1994-12

A friendly editing environment for computer-aided rapid prototyping of hard real-time systems

Rowshanaee, Mehdi E.

Monterey, California. Naval Postgraduate School

NAVAL POSTGRADUATE SCHOOL
Monterey, California

THESIS

A FRIENDLY EDITING ENVIRONMENT FOR COMPUTER-AIDED RAPID PROTOTYPING OF HARD REAL-TIME SYSTEMS

by

Mehdi E. Rowshanaee

December 1994

Thesis Advisor:	Man-Tak Shing
Second Reader:	David Gaitros

Approved for public release; distribution is unlimited

REPORT DOCUMENTATION PAGE

Form Approved
OMB No. 0704-0188

1. AGENCY USE ONLY (Leave Blank)	2. REPORT DATE	3. REPORT TYPE AND DATES COVERED
	December 1994	Master's Thesis

4. TITLE AND SUBTITLE
A FRIENDLY EDITING ENVIRONMENT FOR COMPUTER-AIDED RAPID PROTOTYPING OF HARD REAL-TIME SYSTEMS

5. FUNDING NUMBERS

6. AUTHOR(S)
Rowshanaee, Mehdi, E.

7. PERFORMING ORGANIZATION NAME(S) AND ADDRESS(ES)
Naval Postgraduate School
Monterey, CA 93943-5000

8. PERFORMING ORGANIZATION REPORT NUMBER

9. SPONSORING/ MONITORING AGENCY NAME(S) AND ADDRESS(ES)

10. SPONSORING/ MONITORING AGENCY REPORT NUMBER

11. SUPPLEMENTARY NOTES
The views expressed in this thesis are those of the author and do not reflect the official policy or position of the Department of Defense or the United States Government.

12a. DISTRIBUTION / AVAILABILITY STATEMENT
Approved for public release; distribution is unlimited.

12b. DISTRIBUTION CODE

13. ABSTRACT *(Maximum 200 words)*

The Computer Aided Prototyping System (CAPS) was created to make the software development cycle more efficient. CAPS provides a graphics editor and a syntax-directed editor (SDE) for users to enter prototypes' specifications. The major problem: the user must have a good knowledge of the Prototyping Systems Description Language (PSDL) program syntax in order to use the SDE. Such a requirement imposes a very steep learning curve on a new CAPS user.

The challenge was to minimize the time a user needs to spend in order to complete a development without extensive knowledge of other tools like PSDL and SDE.

The solution is to redesign and implement a new graphics editor to allow the user to enter all PSDL specifications via the graphics editor with hierarchical pull-down menus. The approach taken was to first determine where to start and what type of language and graphic tools should be used in combination to give the ultimate results. There choices were TAEplus, Motif, and the Idraw from Stanford University. Idraw was chosen because of its design, capabilities, and user friendliness.

The major contribution of this thesis is a powerful and user friendly graphics editor to rapidly construct a prototype of a large real-time system. The new CAPS graphic editor has been evaluated by members of the CAPS user group and early feedback shows that the new editor is a truly usable tool. SDE can now take advantage of this graphics editor to enhance the prototyping capabilities of CAPS.

14. SUBJECT TERMS	15. NUMBER OF PAGES
Prototyping Language, Rapid Prototyping, Graphic Editors, Computer Aided Prototyping System, User Interface Toolkits, Real-Time System	342
	16. PRICE CODE

17. SECURITY CLASSIFICATION OF REPORT	18. SECURITY CLASSIFICATION OF THIS PAGE	19. SECURITY CLASSIFICATION OF ABSTRACT	20. LIMITATION OF ABSTRACT
Unclassified	Unclassified	Unclassified	UL

Approved for public release; distribution is unlimited.

A FRIENDLY EDITING ENVIRONMENT FOR COMPUTER-AIDED RAPID PROTOTYPING OF HARD REAL-TIME SYSTEMS

Mehdi E. Rowshanaee
Capitain, United States Army
B.S.E.E., University Of North Dakota, 1986

Submitted in partial fulfillment of the
requirements for the degree of

MASTER OF SCIENCE IN COMPUTER SCIENCE

from the

NAVAL POSTGRADUATE SCHOOL
December 1994

Author: [redacted]
Mehdi E. Rowshanaee

Approved By: [redacted]
Man-Tak Shing, Thesis Advisor

[redacted]
Lt.Colonel David Gaitros, Second Reader

[redacted]
Ted Lewis, Chairman,
Department of Computer Science

ABSTRACT

The Computer Aided Prototyping System (CAPS) was created to make the software development cycle more efficient. CAPS provides a graphics editor and a syntax-directed editor (SDE) for users to enter prototypes' specifications. The major problem: the user must have a good knowledge of the Prototyping Systems Description Language (PSDL) program syntax in order to use the SDE. Such a requirement imposes a very steep learning curve on a new CAPS user.

The challenge was to minimize the time a user needs to spend in order to complete a development without extensive knowledge of other tools like PSDL and SDE.

The solution is to redesign and implement a new graphics editor to allow the user to enter all PSDL specifications via graphics editor with hierarchical pull-down menus. The approach taken was to first determine where to start and what type of language and graphic tools should be used in combination to give the ultimate results. There choices were TAEplus, Motif, and the Idraw from Stanford University. Idraw was chosen because of its design, capabilities, and user friendliness.

The major contribution of this thesis is a powerful and user friendly graphics editor to rapidly construct a prototype of a large real-time system. The new CAPS graphics editor has been evaluated by members of the CAPS user group and early feedback shows that the new editor is a truly usable tool. SDE can now take advantage of this graphics editor to enhance the prototyping capabilities of CAPS.

TABLE OF CONTENTS

ACKNOWLEDGEMENTS

First of all I say thanks to my friend, "The All MIGHTY".

Then I say thanks to the following:

1. Dr. Mantak Shing. For his guidance, understanding, and help.

2. Lt. Chris Eagle. The CODE MASTER that without his knowledge of C++ life would have been very difficult for me.

3. The United States Army, SIGNAL CORPS

I. INTRODUCTION

A. GENERAL

The purpose of this thesis is to design and build a better graphics editor to have a friendly editing environment for Computer Aided Rapid Prototyping Systems (CAPS) to support the rapid prototyping of hard real time systems. First, explanation is needed to clear what is CAPS.

The Computer Aided Prototyping System (CAPS) is a software engineering environment for developing prototypes of real-time systems. It is useful for requirements analysis, feasibility studies, and the design of large embedded systems. CAPS is based on the Prototyping System Description Language (PSDL) [LBY88], which provides facilities for modeling, timing, and control constraints within a software system. CAPS is a collection of tools, implemented in the form of an integrated environment, linked together by a user-interface. CAPS provides the following kinds of support to the prototype designer:

- timing feasibility checking via the scheduler,
- consistency checking and some automated assistance for project planning, scheduling, designer task assignment, and project completion date estimation via the Evolution Control System,
- design completion via the editors, and
- computer-aided software reuse via the software base.

A CAPS prototype is initially built as an augmented data flow diagram and a corresponding PSDL program. The CAPS data flow diagram and PSDL program are augmented with timing and control constraint information. This timing and control constraint information is used to model the functional and real-time aspects of the prototype. The CAPS environment provides all of the necessary tools for engineers to quickly develop, analyze and refine real-time software systems. The general structure of CAPS is shown in Figure 1.

As Figure 1 indicates, CAPS is a collection of tools, integrated by the user-interface. The CAPS User-Interface provides access to all the CAPS tools and facilitates communication between tools when necessary. The tools in the figure are grouped into four sections, Editors, Execution Support, Project Control and Software Base. Each CAPS tool is associated with a different aspect of the CAPS prototyping process.

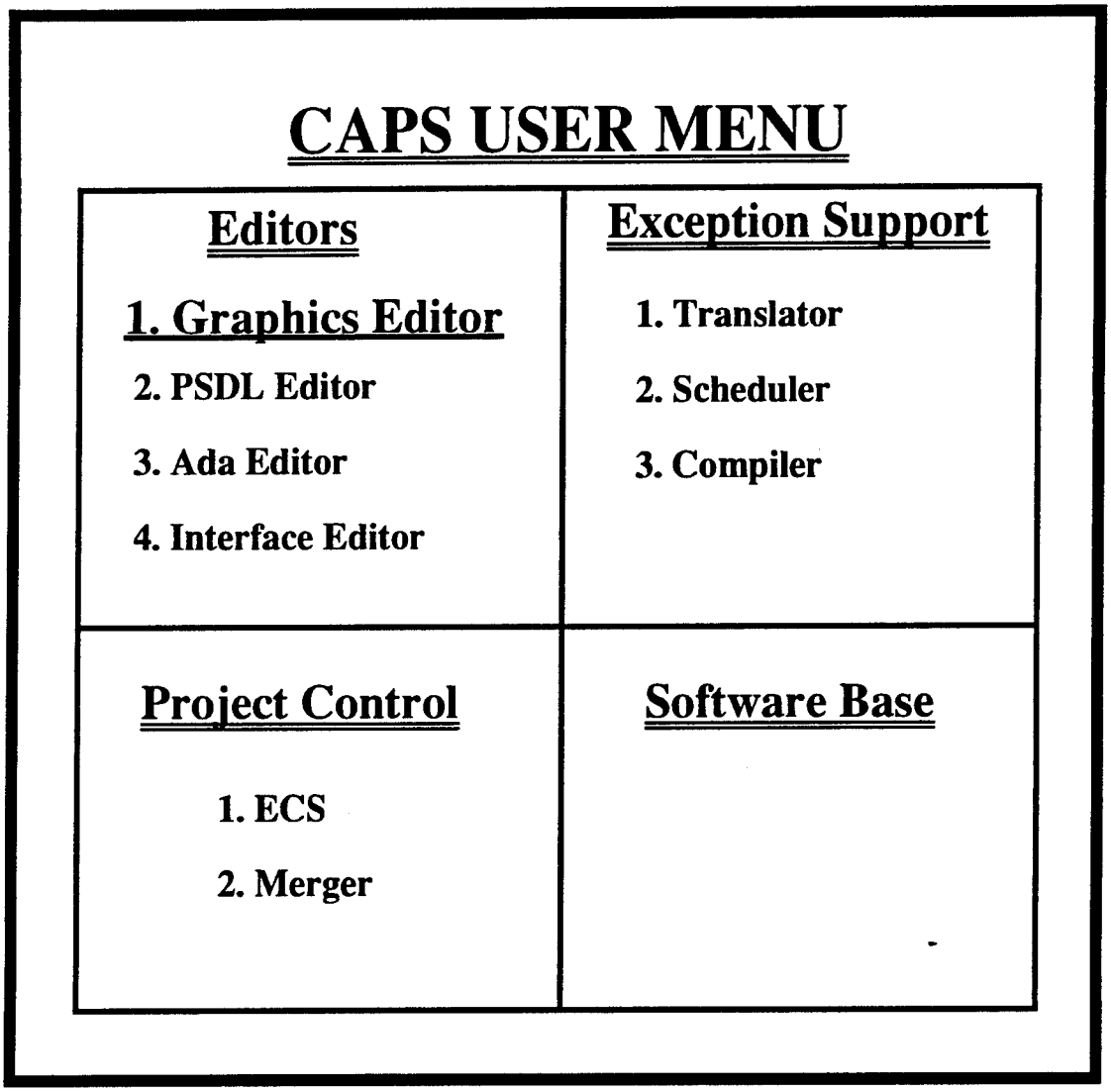

Figure 1.1: The CAPS Development Environment

Rapid prototyping has been presented as an alternative paradigm for software development and evolution. The purpose of prototyping is ensure that proposed

requirements and system concepts adequately match the needs of the prospective clients before detailed optimization and implementation efforts begin. "Computer-aided rapid prototyping improves the efficiency and accuracy of evolutionary development by introducing software tools that assist the designer in constructing and executing the prototype quickly and systematically" [LUQI89]. The general software development cycle with respect to prototyping is shown in Figure 2.

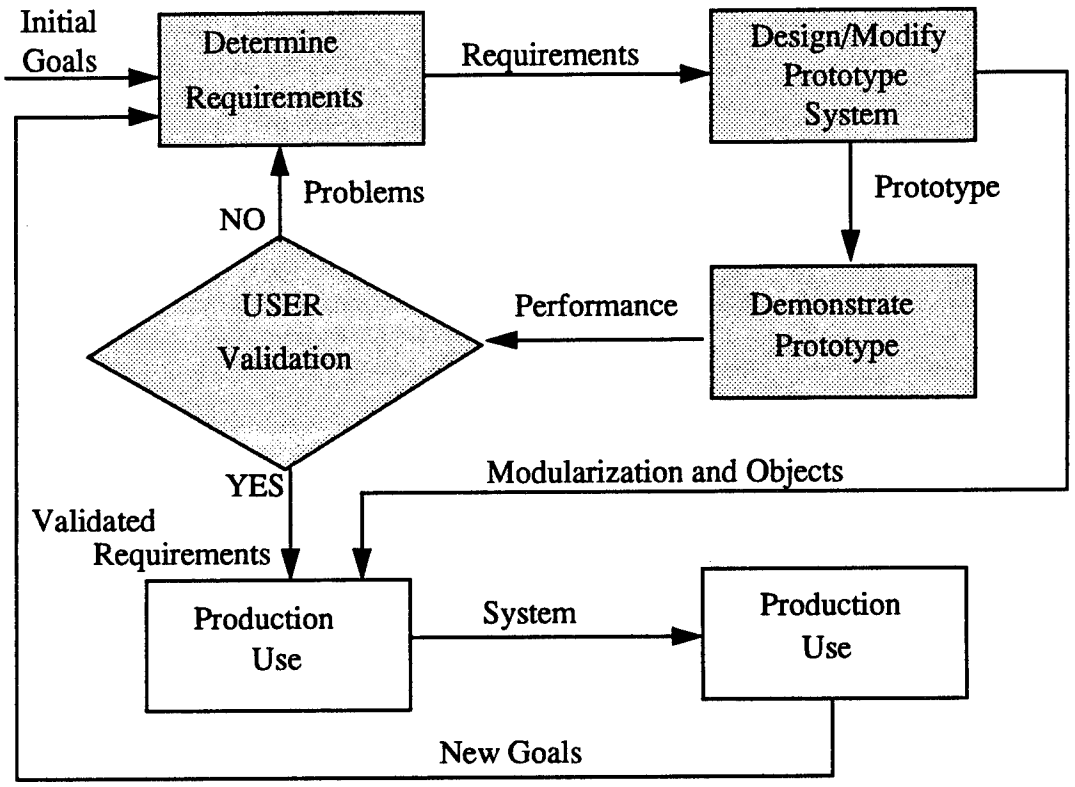

Figure 1.2: The Prototyping Process

CAPS is specifically designed to assist and partially automate the development efforts which lie in the shaded regions of the prototyping process figure. Specifically, based on a set of initial requirements, CAPS allows the engineer to design, modify, demonstrate and validate a software system. Through this process, system requirements can be refined and modified as necessary.

A more precise description of the CAPS prototyping process can be found in the CAPS tutorial [BROC94], and is repeated here for the convenience of the readers. The CAPS prototyping process can be roughly divided into 10 steps listed below. The 10 enumerated steps are accomplished through use of the CAPS tools.

1) Based on requirements, design (or modify) the data flow diagram for the system.

2) Assign all appropriate timing and control constraints to the prototype operators. Assign latencies to data streams (if required).

3) Assign data types to all data streams.

4) Find (in the software base) or build an implementation module for each user-defined data type and each atomic operator. Modules taken from the software base can be modified after retrieval to suit individual needs.

5) Build the prototype's user-interface (if required).

6) Translate the CAPS-generated (and user-augmented) PSDL program into (a portion of) the Ada supervisor module.

7) Run the CAPS scheduler to generate the static and dynamic schedules. This completes the prototype's Ada supervisor module.

8) Compile the prototype.

9) Execute, evaluate and modify (if appropriate) the prototype and/or the requirements.

10) Return to Step 1 if prototype modification is required.

B. PROBLEM STATEMENT

CAPS has been the focus of research efforts for several years. Many different parts of CAPS have been implemented as the end product of Masters' thesis over the years, during different stages of the design effort, using system models with higher level of maturity as time passes.

The focus of this thesis is to create a graphics editor that allows user to annotate the graph with constraint information via menus and buttons graphically, without the need of editing the entire PSDL program in textual form.

C. SCOPE

The scope of this thesis was originally limited to complete the implementation of the last versions of the graphics editor to handle a full-blown PSDL prototype specification. After researching the capabilities of the graphics editor, it became apparent that some of the needed functionality that was included in the CAPS92s' graphics editor was not implemented in the CAPS93s' graphics editor.

First this thesis contains the research to find out what version of the graphics editor should be upgraded in order to achieve maximum user friendliness and to take advantage of the accompanying syntax-directed editing feasibilities. Then after deciding on which version should be upgraded, the other capabilities would be added.

D. ORGANIZATION OF THESIS

Chapter II provides the background of CAPS. Chapter III describes the advantage and disadvantages of each graphics editors' environment and why the choice of upgrading the CAPS92s' graphics editor was made for CAPS94 version. Chapter IV covers lack of documentation and problems encountered. Chapter V is the user manual, and Chapter VI is conclusions and recommendations.

II. BACKGROUND

A. GENERAL

The Computer Aided Prototyping System (CAPS) is a software engineering environment designed to take specifications expressed in the Prototype System Description Language (PSDL) and make them executable. The newest version of CAPS provides an integrated editing environment that requires the user to manually adjust the location of the bubbles, labels, streams, and variables, and to enter the annotated text via the syntax directed PSDL editor. It is desirable to the CAPS users that all the text entries adjustments and modifications be done graphically via hierarchy menus.

One constraint on the graphics part of the CAPS is that it must be implemented in the X Window system, from the Massachusetts Institute of Technology (MIT). This constraint is motivated by portability concerns. The best way of implementing an X Windows-based graphics editor is to use one of the existing toolkits.

During the process of this thesis three toolkits were evaluated:

- The TAE Plus(Transportable Applications Environments Plus) from NASA.
- The Motif toolkit from Open Software Foundation.
- The InterViews toolkit from Stanford University.

This chapter provides a general introduction to these issues.

B. PROTOTYPING SYSTEM DESCRIPTION LANGUAGE

The Prototyping System Description Language (PSDL) is a text-based language designed to express the specifications of real-time systems. It is based on a graphic model of vertices and edges, where the vertices represent *operators*, or software process, and the edges represent the conceptual "flow" of data from one operator to another. Each vertex and edge may have associated timing constraint, and the vertices may have associated control constraints.

Formally, the model used is that of an augmented graph,

$G = (V, E, T(v), C(v))$

where G is the graph, V is the set of vertices, E is the set of edges, $T(v)$ represents the timing constraints for the vertices, and $C(v)$ represents the control constraints for the vertices [LUQI88].

Conceptually, PSDL operators may contain other operators to support the principle of abstraction. An atomic operator is one that is implemented in a programming language, vice a composite operator consisting of other operators and streams.

For example, the following diagram shows a PSDL prototype:

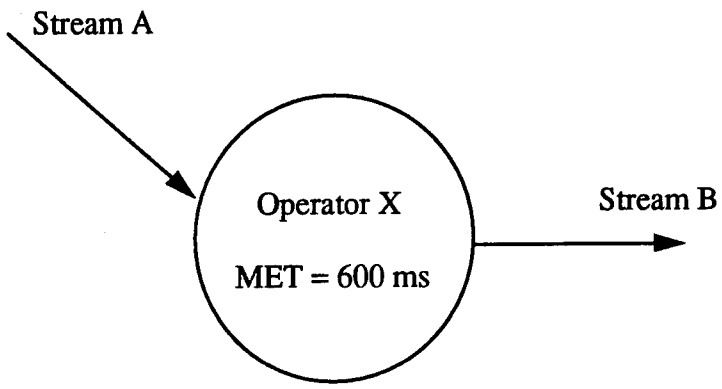

Figure 2.1: Example of PSDL Graph

This graph represents an operation modelled by the Operator X that accepts one item from Stream A, it performs some operation on the data, and outputs Stream B. The Maximum Execution Time (MET), this is the maximum possible time the operator may take to execute the task, defined as 600 milliseconds.

In this example, Operator X is decomposed as follows:

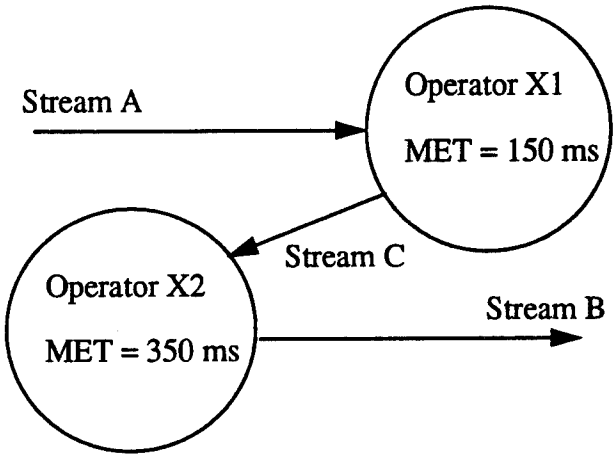

Figure 2.2: Decomposed PSDL Graph

Operator X is a composite operator, while Operator X1 and Operator X2 are atomic operators, implemented in Ada or some other language. The timing and control constraints on these atomic operators must be consistent with those of their parent operator. In a single processor architecture, the combined METs of these atomic operators can not be greater than their parent. Operator X is really not needed to implement the prototype. It serves merely to abstract the functionality of its child operators.

A more detailed description of the PSDL can be found in [LUQI88] and [LUQI89].

C. THE COMPUTER AIDED PROTOTYPING SYSTEM

The Computer Aided Prototyping System (CAPS) is a set of software tools that provides the environment to rapidly build and execute a prototype. The designer of a software system uses a graphics editor to create a graphic representation of the proposed system. The graphic representation is used to generate part of an executable description of the proposed system, represented in the protyping language. This description is used to search a database of reusable software components to find components to match the specification of the prototype. A transformation schema is used to transform the prototype into a programming language. The prototype is then compiled and executed. The end user of the proposed system will evaluate the prototype's behavior against the expected

behavior. If the comparison results are not satisfactory, the designer will modify the prototype and the user will evaluate the prototype again. This process will continue until the user agrees that the prototype meets the requirements. The software designer would be able to see whether the PSDL prototype was logically consistent and semantically and syntactically correct. It would also allow the designer to verify whether the assigned constraints were achievable, or whether either looser constraints or a more efficient implementation were required.

The CAPS was created with these goals in mind[LUQI88]. It is designed to provide the following features to the user:

- automated support for PSDL text editing
- a drawing editor for PSDL graphs
- generation of schedules and control code
- a run-time environment to monitor the execution of prototype
- a data base of reusable components for implementing atomic operators
- a sophisticated engineering data base for managing prototypes

D. INTERVIEWS:A USER INTERFACE TOOLKIT

InterViews is a C++ graphical interface toolkit developed at Stanford University[STAN91]. It includes a library of predefined interactive objects. These objects can be combined to construct many types of user interfaces. InterViews' object oriented approach provides a natural way to build user interfaces as communicating objects.

InterViews runs on the top of X Windows. X Window system was developed to:

- Solve a common problem in large computer networks.
- It was also invented to combine graphics and text easily.
- Display graphics on many different hardware platforms.

In the X Window System, a program is logically separated from the physical display that shows its output. The program is called a client, while the physical unit is called a server.

InterViews provides three classes of objects: interactors, graphics, and text. An interactor class manages some area of potential input and output on a workstation display. All user interface objects are inherited from this class. Example of interactors are bottons, menus, and dialogue boxes. Figure 2.3 shows the hierarchial structure of the interactor class.

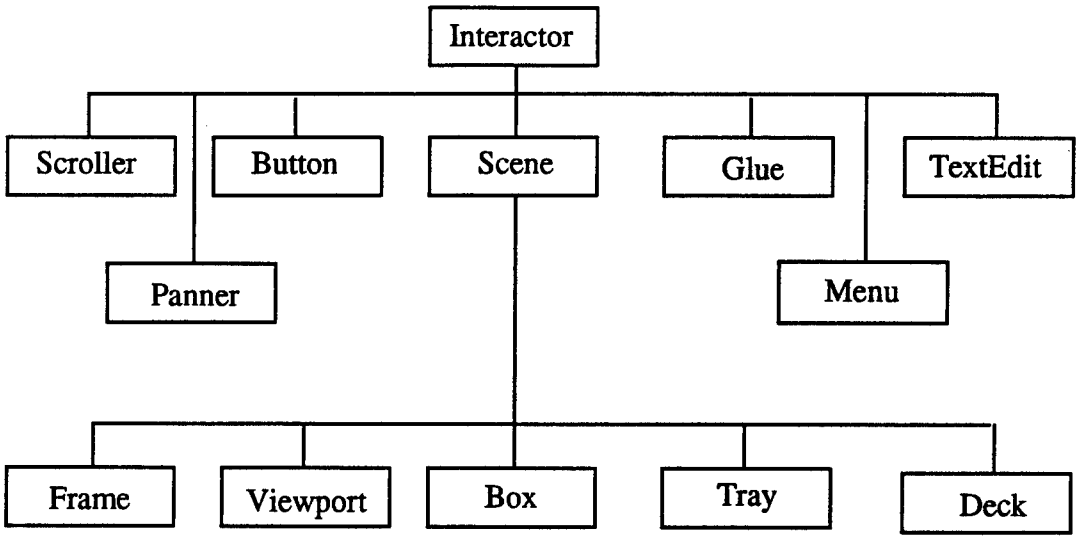

Figure 2.3: Interactor Class Hierarchy

The graphics class defines structured graphics objects. Objects in this class can draw and erase themselves. A state attached to each object which includes attributes of color, line style, and coordinate transformation. Figure 2.4 shows the hierarchical structure of the graphics class.

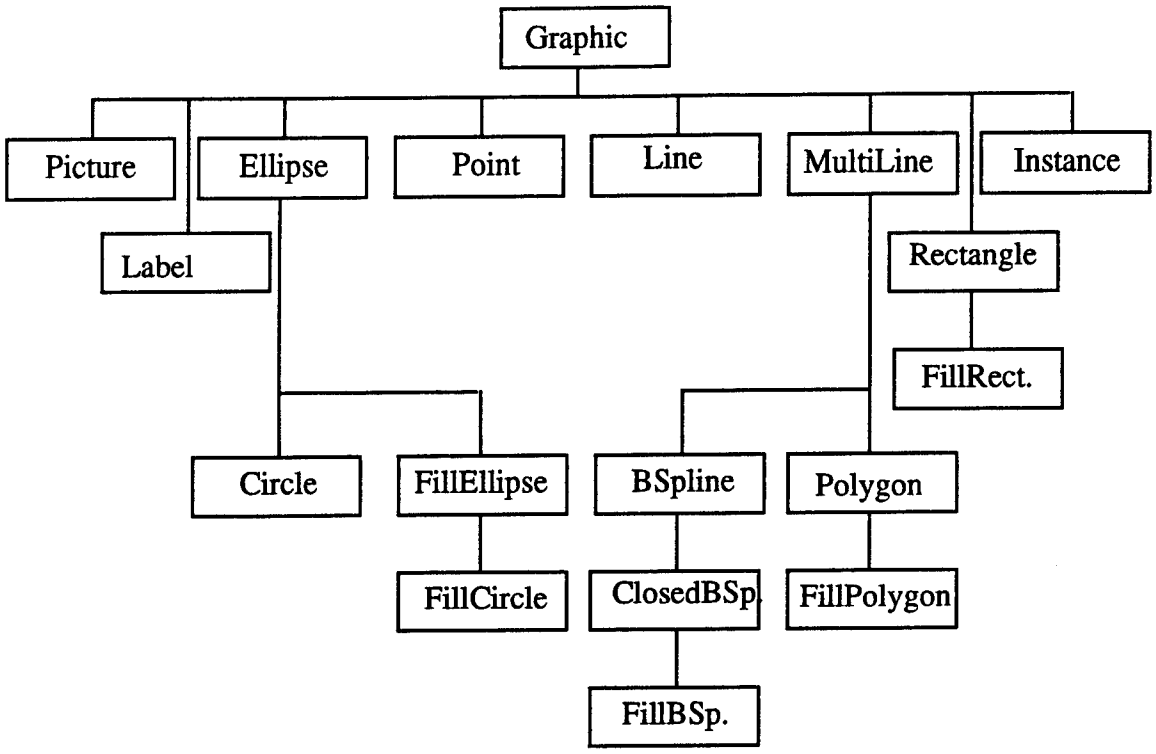

Figure 2.4: Graphics Class Hierarchy

The text class defines structured text objects like strings, text editors, and text buffers.

E. IDRAW: INTERVIEWS DRAWING EDITOR

John Vlissides who is the author of Idraw and Unidraw, used to work for Stanford University, and at the present time he is working for IBM. He used the Interviews toolkit to create a higher-level toolkit designed for the creation of domain-specific graphics editors[VLIS89]. By using a standard set of atomic and composite tools, Idraw is capable of creating an extremely complex graphics editor with a minimum of source code.

Idraw is an object oriented drawing editor provided with InterViews. Idraw provides immediate feedback as the user creates and implements graphic objects such as circles, squares, ellipses, lines, and splines. A picture of Idraw (InterViews' Drawing Editor) is shown in Figure 2.5. Idraw's user interface is composed of InterViews' interactor

and graphics classes. It contains a set of tools on the left side of the editor and a set of pull down menus along the top. The tools are used to create the graphic objects or to change the user's view of the drawing. In order to change a graphic object's attributes, the menus must be used.

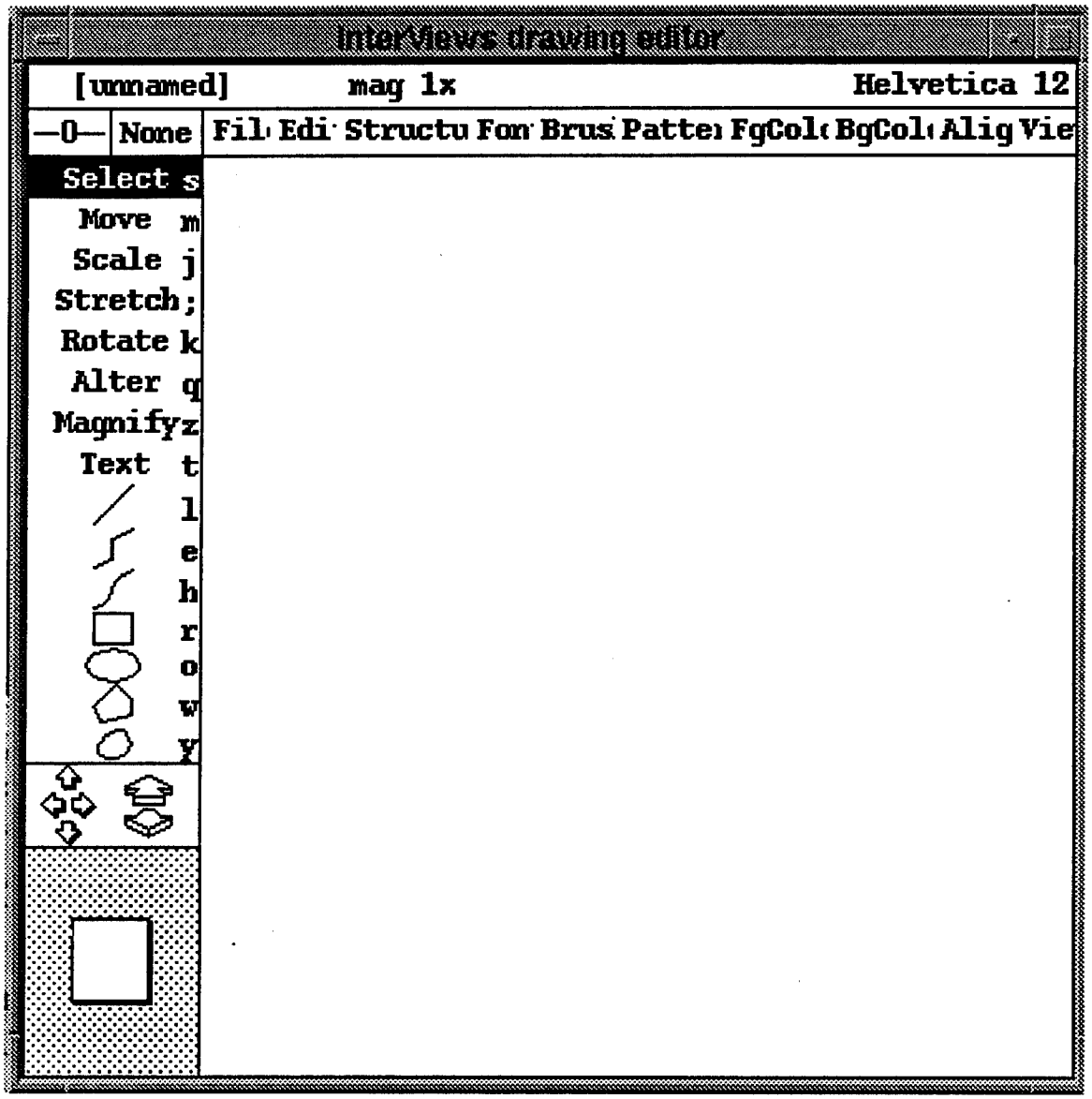

Figure 2.5: Idraw: InterViews Drawing Editor

Idraw allows the user to create the following objects:

- line
- multi line
- open spline
- rectangle
- ellipse
- closed spline
- polygon
- text

Once drawn, these objects can be moved, scaled, stretched, or rehashed. Any part of the drawing can be modified. Various standard editing functions, such as printing a drawing, saving a drawing, and deleting an object, are provided. Certain attributes of the editor, such as font types, brush type, pattern, and foreground and background color can be customized by setting X resources. These customizations are stored with the drawing so not only can be restored when the drawing is edited, but also they are easily retrieved or transported. The drawings are stored in a modified PostScript format.

F. MOTIF

Another toolkit available for X-Window user interface is the Open Software Foundation's Motif toolkit. The Open Software Foundation (OSF) is a consortium of companies including Hewlett-Packard, Digital Equipment Corporation, IBM, and others, whose main business is to enhance the interoperability between computers of different manufacturers.

Motif is a lower-level toolkit than Interviews, providing user interface objects known as widgets with a specific appearance and function. The programming environment provided by Motif is commonly a mix between native X, an intermediate level known as Xt Intrinsics, and higher-level Motif. The Programmer can thus use the pre-defined Motif objects to provide a user interface with a highly standardized appearance, while at the same

time having access to low-level X functions to implement more specific control over the application.

The Motif environment is highly dependent on external configuration options known as resources that allow the installed site and the user a great deal of control over the appearance and functioning of Motif programs. If the programmer so desires, almost every aspect of a Motif program from the color to menu options to fonts can be configured in resource files, allowing the users extreme flexibility in the appearance and function of a program. Resources are modified by editing simple text strings, without the need to recompile the original source code[HELL91].

G. TAE PLUS

The TAE Plus architecture is an extension of the original TAE application management system, called "TAE Classic." TAE Classic is based on a total separation of the user interaction where all user dialogue is directed through an application executive, called the Terminal Monitor (TM). This central control of the user interface provides a consistent look and feel across an application but is limited to ASCII (alphanumeric) terminals.

The advent of graphic workstations led to the development of more elaborate user interface and a closer relationship between the application and the user interface. This often complicated development efforts and particularly the porting of applications to different hardware platforms. TAE Plus addresses the needs of the application developer of graphical user interfaces by providing an effective and portable mechanism for separating the interface and application as much as possible.

The current release of TAE Plus is designed to be portable across a wide range of machines and operating systems. TAE Plus is designed to run on UNIX and DEC VMS systems that support the X Window System. The user interface looks and feel is based on the Open Software Foundation's Motif specification for graphical user interface.

The TAE Plus WorkBench is a development tool that supports the interactive design and layout of graphical user interfaces. The major elements of a user interface are

"panels" and "interaction items." TAE Plus initial WorkBench file selection menu is shown in Figure 2.6 below, and the actual TAE Plus WorkBench is shown on Figure 2.7.

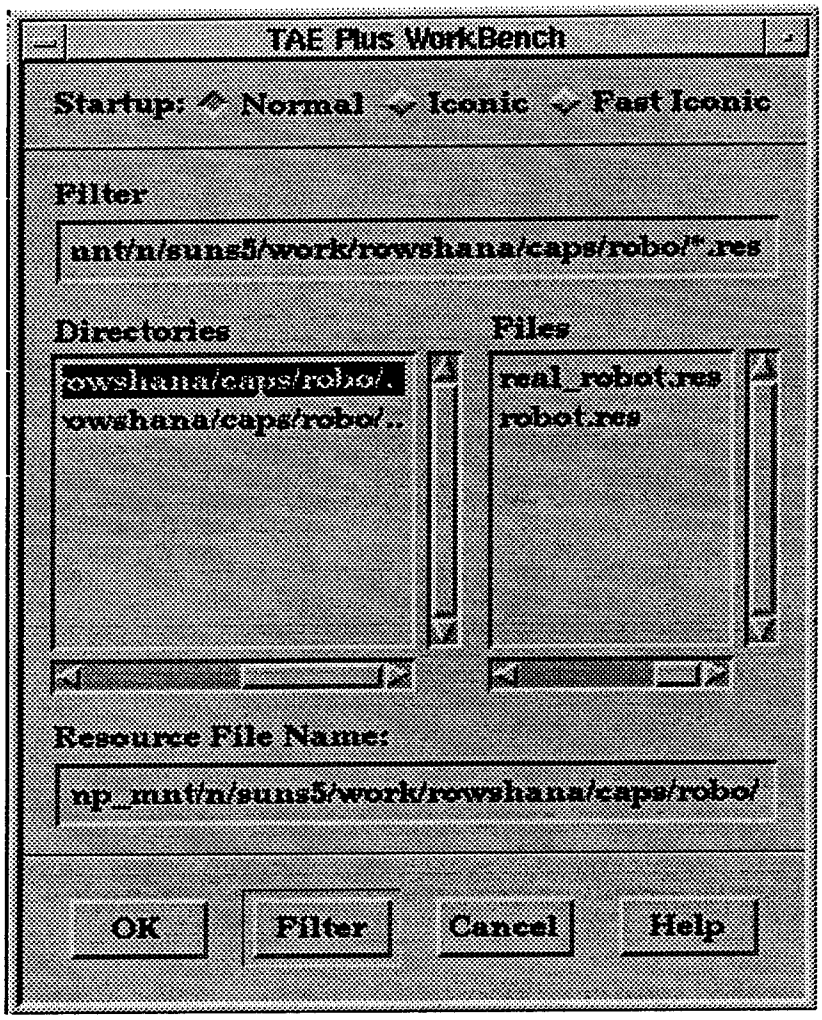

Figure 2.6: TAE Plus WorkBench file Selection Menu

A TAE Plus panel is similar to an individual X window, and is used to hold a collection of interaction items, as it can be seen in TAE Plus WorkBench below.

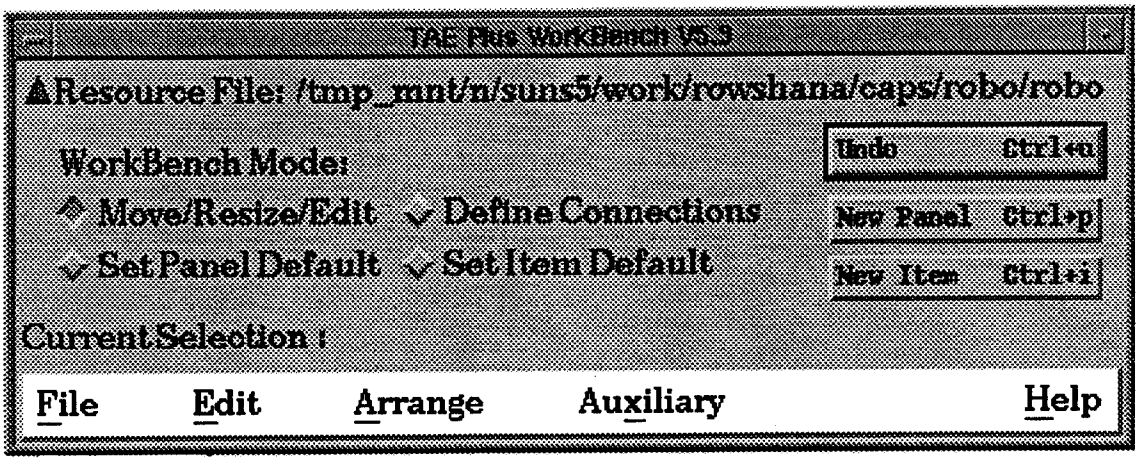

Figure 2.7: TAE Plus WorkBench

TAE Plus interaction items receive information from, and display information to, the user. They include such objects as buttons, scrolling lists, pulldown menus, text input fields, and dials. With the WorkBench, a user interface developer can:

• Define, resize, move, copy, and set the display attributes and default values of interaction items and panels.

• Define user interface connections that associate the user selection of an interaction item, such as button, with a change in state of that button's panel and/or a change in state of another panel.

• Rehearse a user interface, trying out defined connections, and reviewing the look and feel of the interface from the end user's perspective.

• Generate application code that displays and controls the designed user interface.

• Save the user interface separately from the application code in a TAE Plus resource file.

• Modify an existing user interface, with few or no changes required to the application code.

So after investigating the prospects of using TAE Plus for CAPS graphics editor, and testing the TAE Plus, I noticed that they use the InterViews' Idraw for their graphics editor, Figure 2.8 shows the TAE Plus's graphics editor. So, concluded that this is not the

right choice. TAE Plus is a very powerful tool to make interfaces, but not creating a graphics editor for CAPS purpose. On the other hand it is obvious that because TAE Plus is using InterViews' Idraw for their graphics editor, Idraw must be a powerful tool.

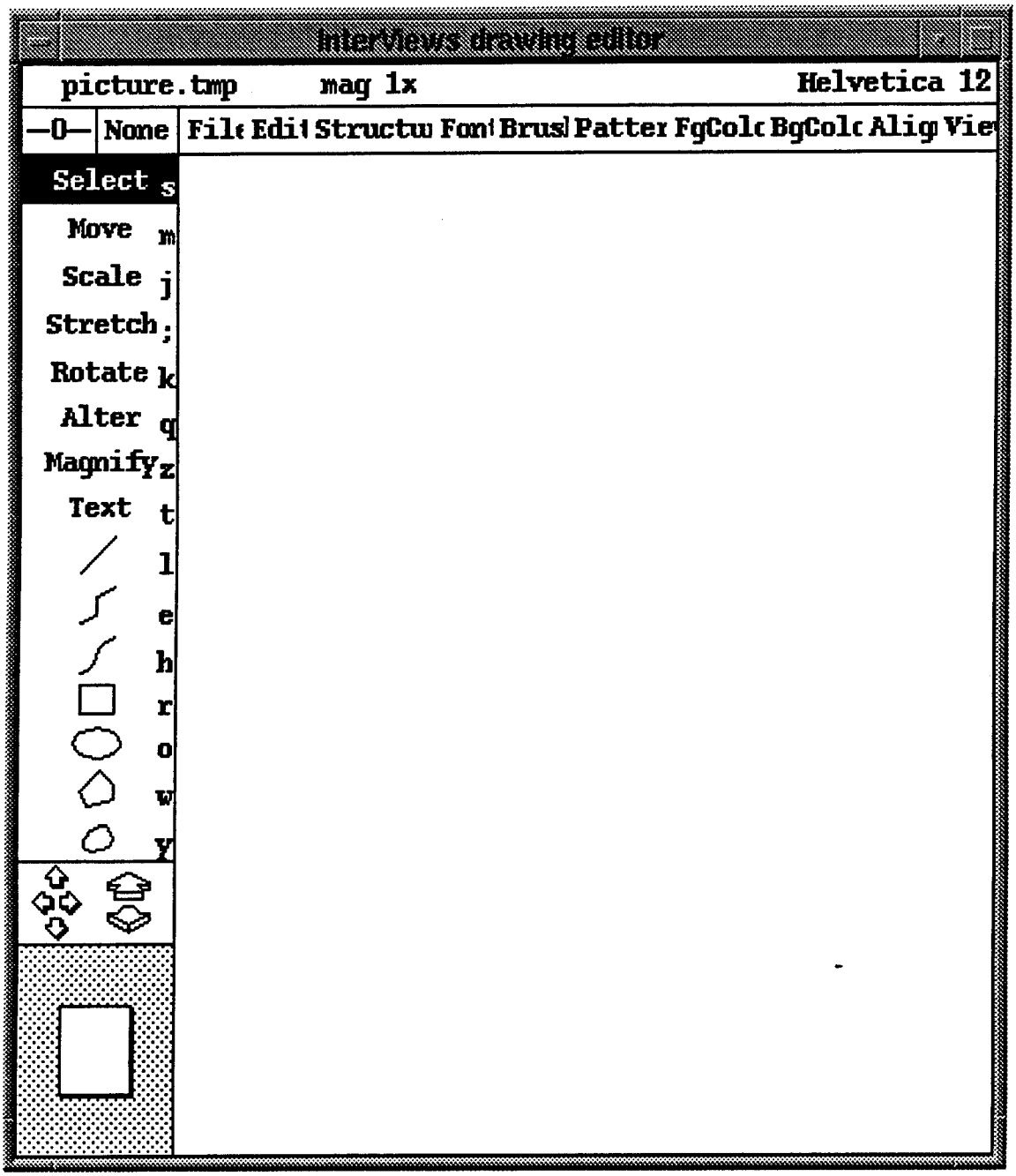

Figure 2.8: TAE Plus's Graphics Editor

III. DESIGN OF THE CAPS'94 GRAPHICS EDITOR

A. INTRODUCTION

As stated in the introduction, the focus of this thesis is the Graphics Editor, which is the user interface for building prototypes. The existing graphics editors of versions CAPS92 and CAPS93 are completely different. CAPS92 used the InterViews' Idraw for a base [CUMM90]. In CAPS93's a generic text editor based on the Motif toolkit is used to edit all the PSDL functions [DIXO92].

B. CAPS92 GRAPHICS EDITOR

In CAPS92, InterViews' Idraw was chosen as the graphics editor tool because of its powerful graphics, Postscript output, editing capabilities, object oriented representation, and ease of use. It also provides a variety of fonts, colors, brush types and patterns to enhance the drawing. The following is an object oriented analysis of the Idraw.

Class **Idraw**

Idraw is the main class of the editor. It opens a given drawing file, if any, creates the editor on the screen, and calls the event handler, **Run**, to process the user's action until the user chooses to terminate the editor.

Idraw depends on several classes. A diagram of these dependencies is shown in Figure 3.1. The class hierarchy is as follows:

Class **Idraw**

Behavior: Displays a drawing editor.

Set of Attributes:

initial_drawing:	Name of drawing file to edit.
cmds : Commands:	Display a pull down menu bar which contains many pull down menus.
drawing : Drawing:	Performs operations on the drawing.
drawing : DrawingView:	Displays drawing.

editor : Editor:	Handles drawing and editing operations.
mapkey : MapKey:	Maps characters to Interactors.
panner : Panner:	pans and zooms drawing.
state : State:	Stores current state information about drawing.
stateview : StateView:	Displays current state information.
tools : Tools:	Displays drawing Tools.

operations:

Idraw:	Parses command line arguments, initializes attributes, and displays editor.
Run:	Opens drawing file, and processes user's choices until the editor is terminated.

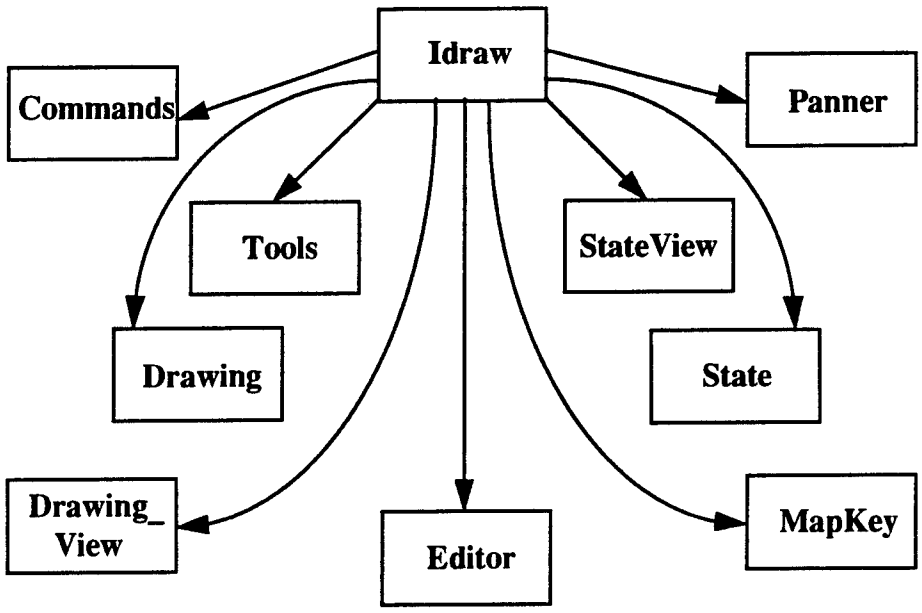

Figure 3.1: Idraw Dependency Diagram

Class **Commands**

Commands provides over 100 traditional editing commands. It is displaying as a menu bar containing a collection of pull down menus referenced by the title of the menu. The following groups of editing functions are represented by the menus:

File: Presents file operations such as opening file, saving a file, and printing a file.

Edit: Presents editing operations such as deleting a selected object, cutting an object from the drawing, and pasting a cut object in the drawing.

Structure: Presents operations to change the way objects are structured together such as grouping selected objects together, bringing an object to the front of the group, and sending an object to the back of the group.

Font: Presents a list of fonts to be used for the text in the drawing.

Brush: Presents various types of brushes such as solid lines, dashed lines, and directed lines anchored at one end.

Pattern: Presents a list of patterns for the selected graphic objects.

Color: Presents a list of foreground and background colors for the selected graphic objects.

Align: Presents various alignment options such as aligning left sides of selected objects, and aligning centers of selected objects.

Options: Presents various options to aid in putting the drawing on the page, such as reducing a drawing to fit on one page, centering a drawing to the page, and providing a grid.

Class **Drawing**

The **Drawing** contains the internal representation of the picture. The user's drawing is stored as a linked list of graphics objects. The interface to modify the objects is also provided. Drawing depends on two classes:

PictSelection: Linked list of objects in drawing.

SelectionList: Linked list of those objects to be modified by a command or tool.

Class **DrawingView**

The **DrawingView** provides the user's view of the drawing. It is responsible for everything drawn on the screen.

Class **Editor**

The **Editor** performs the operation selected by the user for the given tool or command. It uses **Drawing** to modify the internal representation of the drawing when needed.

Class **MapKey**

Each of the tools and commands can be executed by typing a letter as a shortcut to clicking with the mouse. **MapKey** maps the letter to the tool or command desired. It stores all of the tools and commands in a table indexed by the shortcut letter.

Class **Panner**

The **Panner** is used to pan and zoom the drawing in order to view the drawing close up or farther away.

Class **PictSelection**

Drawing stores all of the graphic objects drawn on the screen in picture, an instance of the **PictSelection** class. Each object is an instance of a class inherited from the class **Selection**. **Selection** is inherited from the InterViews Graphic class. **PictSelection** is inherited from **Selection** and is a linked list of all the **Selections** in the drawing. The Selection inheritance hierarchy is shown in Figure 3.2. The **NPtSelection**, also shown in

the figure, is a special type of graphic object that can draw arrowheads on one or both of its endpoints.

Class **SelectionList**

The **SelectionList** contains a subset of the **Selections** in the drawing. It is a linked list of only those **Selections** chosen by a drawing tool or editing command.

Class **State**

The State stores state information about the user's drawing and paint attributes to be used when creating new graphic objects. Some of the information stored are the brush type, drawing name, font, and pattern.

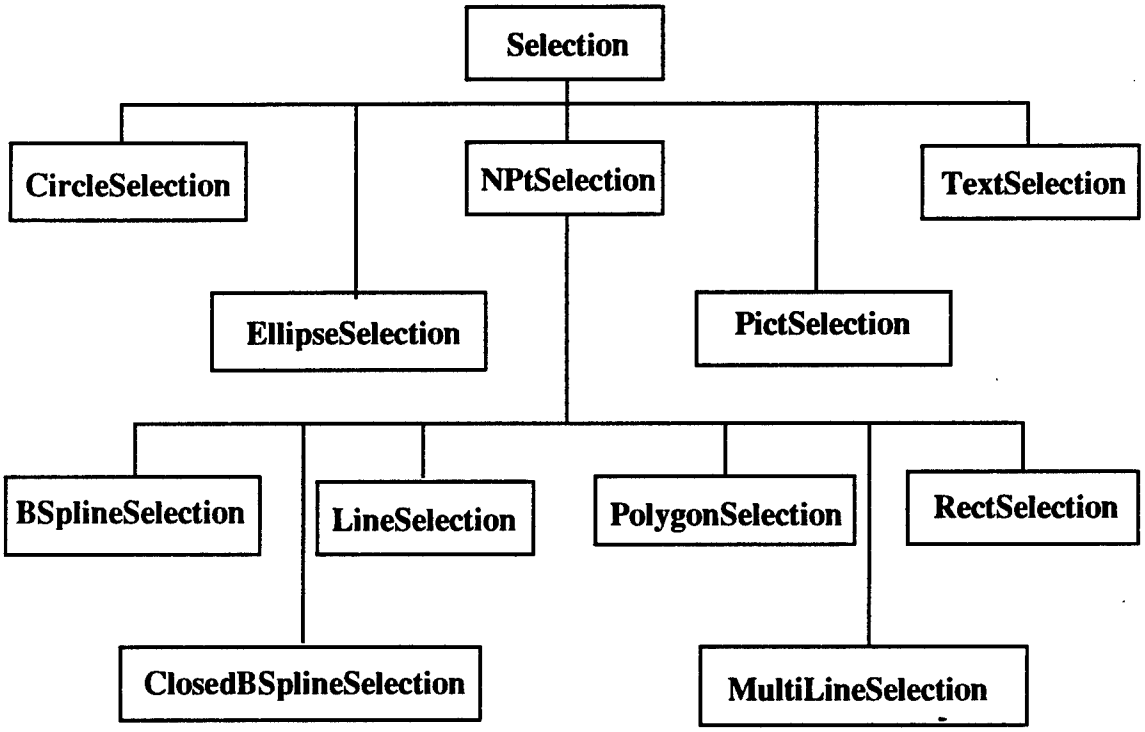

Figure 3.2: Selection Hierarchy

Class **StateView**

Idraw keeps track of and displays current state information. The kinds of information displayed are:

- current brush type

- name of current drawing
- current font
- status of gridding
- magnification percentage
- current modification status of drawing
- current pattern

Each piece of information is a separate object and inherited from **StateView**. The **StateView** hierarchy is shown in Figure 3.3 below.

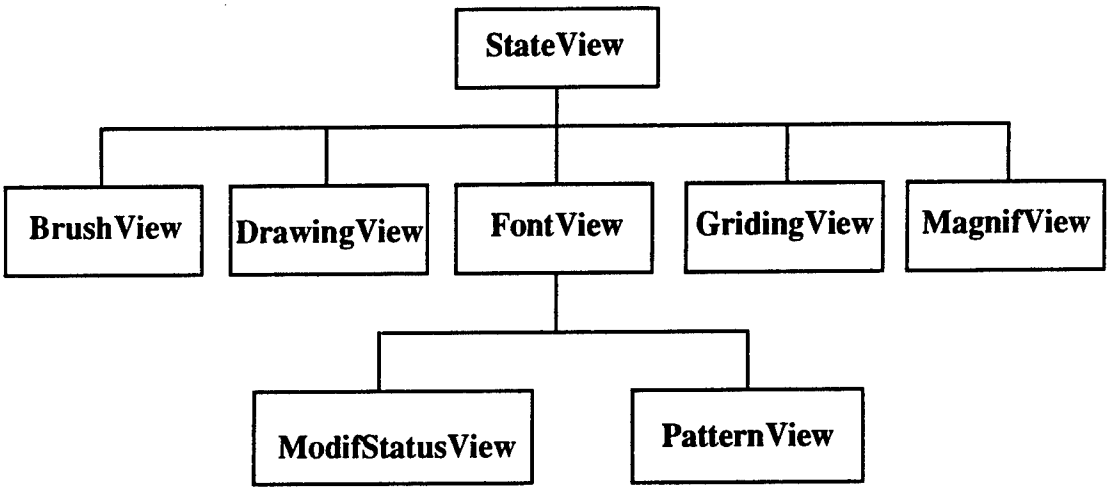

Figure 3.3: StateView Hierarchy

Class **Tools**

Tools creates the panel that displays the drawing tools to the user. These drawing tools are:

- Select one or more graphic objects.
- Move the selected objects to another location on the screen.
- Scale the selected objects.
- Stretch the selected objects.
- Rotate the selected objects.
- Reshape the selected objects.

- Magnify the selected objects.
- Add text to the drawing.
- Draw a line.
- Draw a multiline.
- Draw an open spline.
- Draw an ellipse.
- Draw a rectangle.
- Draw a polygon.
- Draw a closed spline.

The panel created by **Tools** is made up of panel items known as **IdrawTools**. Each of the Tools functions is represented by a class inherited from **IdrawTool**. Figure 3.4 shows hierarchy of these tools.

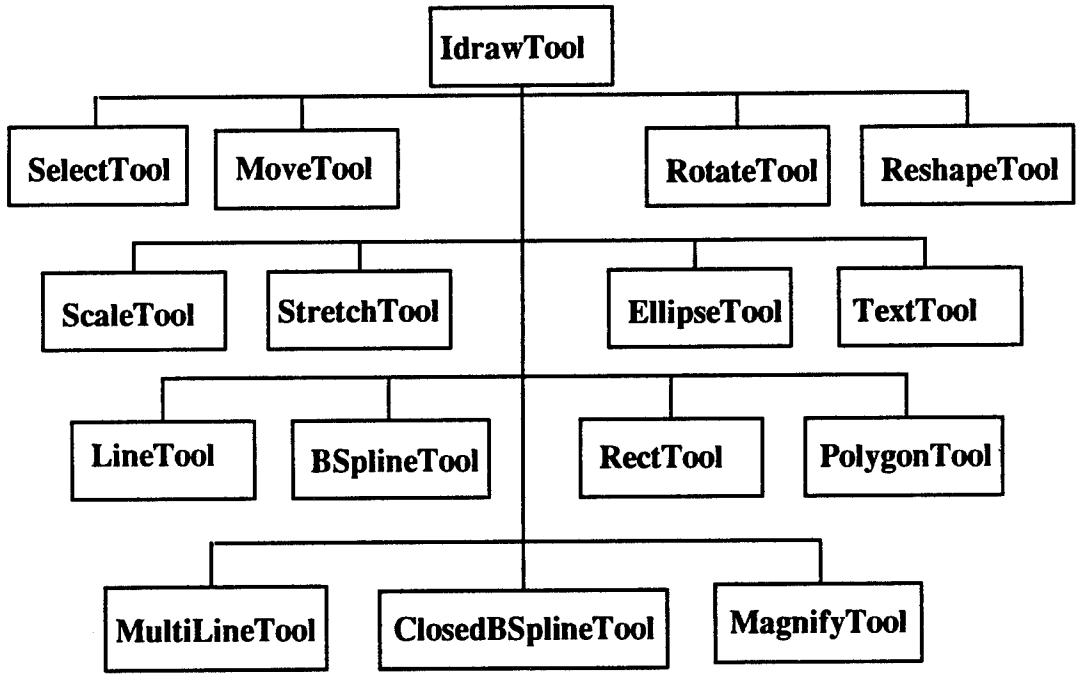

Figure 3.4: IdrawTool Hierarchy

CAPS92 graphics editor was accomplished by making some major changes to the existing Idraw codes. The following is the list of the major changes to Idraw to construct the graphics editor:

- Present Prototype Names To The User.
- Remove Unused Commands and Tools.
- Add Internal Representation of DFD.
- Modify Existing Commands and Tools.
- Add New Tools.
- Add Ability to Rebuild DFD Data Structure.
- Add Message Block.

Detailed information can be found in Mary Ann Cummings' thesis on The Development of User Interface Tools for CAPS, [CUMM90].

C. CAPS93 GRAPHICS EDITOR

Motif was used for creation of CAPS93s' graphics editor and was developed by the Open Software Foundation (OSF), an industry consortium that includes Digital Equipment Corp., Hewlett-Packard, and IBM. Motif actually refers to any one of the following:

- A look-and-feel Style Guide for applications, based on IBM's Common User Access (CUA) guidelines, which are also implemented in OS/2 and Microsoft Windows.
- A window manager, mwm, to help enforce the Style Guide.
- A User Interface Language (UIL) interpreter, which places much of the user interface code into interpreted files.
- A toolkit (C library) for building style Guide-compliant applications, also called a widget set. This toolkit is based on the Xt Intrinsics, as are other toolkits like the Athena widgets and OLIT.

Motif allows you to create programs with a graphical-user interface that can run on a wide variety of computer platforms, including systems from Sun Microsystems, Hewlett-Packard, IBM, Silicon Graphics, DEC, and a slew of clone vendors who support UNIX on the 386/486/Pentium architectures.

There are a number of reasons why we might want to use the Motif toolkit:

• Motif is one of the major interface standards in the UNIX world.

• Motif was adopted as part of the Common Open Software Environment, or COSE, led by IBM, Hewlett-Packard, Sun, SCO, Univel, and UNIX System Laboratories.

• OSF/Motif compliance is needed for selling to many big firms and organizations.

• Using an X toolkit-any X toolkit- speeds programming.

• Motif provides nice-looking 3D effects.

• Motif fits in reasonably well with X standards with the user of window managers and resource files.

• Motif also is part of some of the newer trends in software design and development. Included in this trend is the notion of EVENT-DRIVEN programming.

The CAPS93's graphics editor was intended to implement some very specific behavior that is not directly available under InterViews, for example the UNDO function to undo the recent changes before saving the prototype.

Motif gives the simple capabilities of drawing circles, lines, and rectangles, but it is not a graphics editor. The capabilities that is needed and it is not included in CAPS93's graphics editor, they exist in CAPS92s' graphics editor. These capabilities requires a lot of programming time and are as follows:

- Reduce to fit
- Redraw and Center page
- Grouping of objects
- Moving groups of objects
- Different brush style
- Delete an object or group of objects
- Select All

Idraw is an ideal tool, it has all the needed capabilities to handle all the graphic needs of CAPS. CAPS92s' graphics editor is well capable of many needed functionality for manipulating prototypes data.

D. CONCLUSION OF GRAPHICS EDITOR CHOICE

After extensive investigation and evaluation, the choice is clear. Upgrading the CAPS92s' graphics editor to give the needed capabilities gives a more capable graphics editor to the users of CAPS. Idraw will provide much of the complicated required functionality that is needed. This will save lots of programming time. Reusing existing codes is one of the most important part of Software Engineering. By using the existing codes of Idraw which is a good and bug free code will save lots of time.

E. DESIGN CHOICES OF THE GRAPHICS EDITOR

The CAPS94 graphics editor will have an extra menu bar button by the name of Property. Property will be consisted of the following menu buttons:

- Constraint by.
- Stream.
- Operator Specification.

Constraint by is a menu button which consists of others command buttons and menu buttons. The difference between the command buttons and menu buttons is: the menu buttons will present the user with more command buttons, while the command buttons will activate a procedure to handle the required actions. Under Constraint by menu the command buttons are:

- Output Gaurd.
- Exception Condition.

And the menu buttons are:

- Trigger.
- Finish Within.
- Minimum Calling Period.
- Maximum Response Time.
- Period.
- Timer operator.

Stream button will activate a command button and a menu button and they are:

- Label button.
- Latency menu.

Label button activates a procedure which gives the capability to enter labels for the operators and the streams (vertices and edges). Latency menu will activate another menu with five command buttons. These buttons are the units of time that a stream (edge) may have to enter the time and they are:

1. Hour.
2. Minute.
3. Second.
4. Millisecond.
5. Microsecond.

Operator Specification menu consists of five command buttons and three menu buttons. The command buttons are:

- Key Words.
- States.
- Informal.
- Formal.
- Exception_Declaration

And the menu buttons are:

- Generic.
- Input.
- Output.
- MET.

Key Words, States, Informal, Formal, and the Exception_Declaration will activate the related procedures for proper actions. Generic, Input, and Output menu buttons will activate another menu with four choices of data types and they are:

1. integer.
2. real.
3. boolean.

4. user defined.

MET menu button will give the user five choices of time units and they are:

1. Hour.
2. Minute.
3. Second.
4. Millisecond.
5. Microsecond.

F. DATA STRUCTURE

It is possible to use the existing data structure of CAPS92, and expand it to meet PSDL's grammer requirements. The existing data structure is as follows:

OperatorSelList is doubly linked list of OperatorSelNode. OperatorSelNode's internal pointer points to the Class OperatorSelection. Class OperatorSelection is the following:

```
class OperatorSelection {
public:
    OperatorSelection(EllipseSelection*);
    ~OperatorSelection();
    void SetEllipseSelection(EllipseSelection*);
    EllipseSelection* GetEllipseSelection();
    void SetTextSelection(TextSelection*);
    TextSelection* GetTextSelection();
    void SetMETSelection(TextSelection*);
    TextSelection* GetMETSelection();
    void InputDataFlowSplineAppend(DFDSplineSelNode*);
    void OutputDataFlowSplineAppend(DFDSplineSelNode*);
    void SelfLoopAppend(DFDSplineSelNode*);
    char* FindSplineInList(TextSelection*, BSplineSelection*);
    DFDSplineSelList* GetInputDFSplineList();
```

```
    DFDSplineSelList* GetOutputDFSplineList();
    DFDSplineSelList* GetSelfLoopList();
    boolean FoundSplineInLists(BSplineSelection*, BSplineSelection*);
    boolean FoundSelfLoopInList(BSplineSelection*, BSplineSelection*);
    char* FoundTextInLists(TextSelection*, TextSelection*);
    void FoundSecondTextInLists(char*, TextSelection*);
    char* GetPSDLText();
    boolean FindSplineInSecondList(char*, char*, BSplineSelection*);
     void SetPSDLText(char*);
    boolean AddLatencyToSplineInList(TextSelection*, BSplineSelection*);
    void RemoveInputFromPSDL(TextSelection*);
    void RemoveOutputFromPSDL(TextSelection*);
    void RemoveStateFromPSDL(TextSelection*);
     void SetId(int);
    int GetId();

protected:
    void AddOperatorIdToPSDL(char*);
    void AddInputToPSDL();
    void AddOutputToPSDL();
    void ReplaceInputStringInPSDL(char*, char*);
    void ReplaceOutputStringInPSDL(char*, char*);
    int FindIndex(char**, int);
    void ReplaceStateStringInPSDL(char*, char*);
    void AddStateToPSDL();
    void AddMETToPSDL(TextSelection*);

    int opId;
    EllipseSelection* elsel; .....// operator drawn in DFD
```

```
    TextBuffer* txb; .....// buffer containing PSDL
    TextSelection* txtsel; .....// operator's label
    TextSelection* metsel; .....// operator's MET
    DFDSplineSelList* input_df_spline_list;  // list of input data
     // flow splines
    DFDSplineSelList* output_df_spline_list;  // list of output data
     // flow splines
    DFDSplineSelList* selfloop_list;        // list of self loops
};

inline void OperatorSelection::SetId(int id) {
  opId = id;
}
```

The same procedures can be used with different names to process the additional needed PSDL's data. For example for adding TrrigerByAll the following may be added to the protected part of this class as follow:

```
void AddTriggerByAllPSDL ( TextSelection *);
TextSelection * triggerByAllsel ;.....//operatore's Trriger By All
```

Also the data flow diagram of CAPS92 can be expanded to meet the additional PSDL grammar that is necessary for complition of CAPS' graphics editor. The existing data flow diagram is:

```
// keywords to be inserted when creating PSDL
#define OPER_TKN      "OPERATOR "
#define SPEC_TKN      " SPECIFICATION\n"
#define INPUT_TKN     " INPUT\n"
#define OUTPUT_TKN    " OUTPUT\n"
#define ST_TKN        " STATES\n  UNDEFINED_ID : UNDEFINED_TYPE\n
INITIALLY\n  UNDEFINED_EXPRESSION\n"
#define MET_TKN       " MAXIMUM EXECUTION TIME "
```

```
#define DESC_TKN      " DESCRIPTION "
#define TEXT_TKN      "{ UNDEFINED_TEXT }\n"
#define END_TKN       " END\n"
#define IMP_TKN       " IMPLEMENTATION\n"
#define GR_TKN        " GRAPH\n"
#define VER_TKN       " VERTEX "
#define EDGE_TKN      " EDGE "
#define ID_TKN        " UNDEFINED_ID"
#define EXT_TKN       " EXTERNAL"
#define TYPE_DECL_TKN " UNDEFINED_ID : UNDEFINED_TYPE"
#define IMP_ADA_TKN   " IMPLEMENTATION ADA "
#define STREAM_TKN    " DATA STREAM\n"
#define CON_TKN       " CONTROL CONSTRAINTS\n"
#define CON_OP_TKN    " OPERATOR UNDEFINED_ID\n"

// keywords to be used for search through PSDL text buffer. They are
// different from those above because the text buffer can't ever locate
// newlines
#define INPUT_SCH_TKN  "INPUT"
#define SPEC_SCH_TKN   "SPECIFICATION"
#define OUTPUT_SCH_TKN "OUTPUT"
#define GEN_SCH_TKN    "GENERIC"
#define STATES_SCH_TKN "STATES"
#define EXCEPT_SCH_TKN "EXCEPTIONS"
#define MET_SCH_TKN    "MAXIMUM EXECUTION TIME "
#define MCP_SCH_TKN    "MINIMUM CALLING PERIOD"
#define MRT_SCH_TKN    "MAXIMUM RESPONSE TIME"
#define KEY_SCH_TKN    "KEYWORDS"
#define DESC_SCH_TKN   "DESCRIPTION"
```

#define AX_SCH_TKN "AXIOM"

#define END_SCH_TKN "END"

#define STREAM_SCH_TKN "DATA STREAM"

#define TIMER_SCH_TKN "TIMER"

#define CON_SCH_TKN "CONTROL CONSTRAINTS"

#define MET_SDE "psdl_editor"

#define STREAMS_SDE "psdl_editor"

#define CONSTRAINTS_SDE "psdl_editor"

#define SPECIFICATION_SDE "psdl_editor"

So there exists two kinds of keywords. Those which are used for creation of PSDL have the form XXXXXXX_TKN, and those for PSDL text buffer with the form XXXXXXX_SCH_TKN. In order to meet all the PSDL grammer's requirement the following keywords must be added to the dfd_defs.h file.

- #define TRIGGER_IF_TKN " TRIGGER IF "
- #define TRIGGER_BY_ALL_TKN "TRIGGER BY ALL"
- #define TRIGGER_BY_SOME_TKN "TRIGGER BY SOME "
- #define FINISH_WITHIN_TKN " FINISH WITHIN"
- #define MAX_RES_TIME_TKN " MAXIMUM RESPONSE TIME"
- #define PERIOD_TKN "PERIOD "
- #define INFORMAL_TKN " INFORMAL"
- #define FORMAL_TKN "FORMAL"
- #define OUTPUT_GAURD_TKN "OUTPUT GAURD"
- #define TRIGGER_IF_SCH_TKN " TRIGGER IF "
- #define TRIGGER_BY_ALL_SCH_TKN "TRIGGER BY ALL"
- #define TRIGGER_BY_SOME_SCH_TKN "TRIGGER BY SOME "
- #define FINISH_WITHIN_SCH_TKN " FINISH WITHIN"
- #define MAX_RES_TIME_SCH_TKN "MAXIMUM RESPONSE TIME"
- define PERIOD_SCH_TKN "PERIOD "
- #define INFORMAL_SCH_TKN "INFORMAL"

- #define FORMAL_SCH_TKN "FORMAL"
- #define OUTPUT_GAURD_SCH_TKN "OUTPUT GAURD"

IV. IMPLEMENTATION CHALLENGES

A. LACK OF DOCUMENTATION

Both InterViews and Idraw are research tools developed at Stanford University. As such, the availability of any technical document that relates to any of the modules' codes does not exist. Only few tutorial pages were found after searching the Internet. There was no documentation that explains how the pieces fit together. There is no reference material for the programmers who chose to modify the behavior of any existing tools. The only available form of documentation is most instances is to examine the uncommented library source code for the tool in hopes of figuring out how things work. Of course if the software were written in Ada, it would be very easy to figure each module out. DOD is the only one so far who knows the true power and value of Ada. The software development and research communities often ignore the biggest benefit of Ada in reducing the cost of developing a software is in maintanace, upgrading, and future modifications. It is hard for an experienced programmer and software developer who uses Ada to generate spaghetti codes, but in C or C++ they have to work really hard to develop a clean code that can be followed in the future by other professionals. InterViews and Idraw were written in C++. So, lack of documentation is not the only short coming.

B. PROBLEMS

First problem was encountered during compilation of the CAPS92 source code. Because the graphics editor is using so many different modules from InterViews library and the X-window library, setting up the paths in Makefile took few days with the size of 85757 bytes. Had to figure out the correct versions through trail and error compilations.

Second problem was encountered during the design of the hierarchy menus. InterViews's Idraw does not handle any hierarchy of menus. In order to get the desired functionality a hierarchy menu is the must. Idraw making a sophisticated use of inheritance of C++. Inheritance is a property of object-oriented languages like C++ that allow the programmer to reuse existing software implementations, adding and modifying features in

a structured way. While this is a great convenience for the programmer, it results in some objects being implemented with five to nine levels of inheritance. At the leaf nodes of the inheritance tree, the reader sees methods and variables used, often with no idea where they are defined. Small pieces of code are added at each level of inheritance, spreading the complete definition of some operations up and down the inheritance tree. It was very difficult to figure out what things do without a good documentation, and it took in excess of six weeks to find out where each module is coming from and what they do. This is true example of perfect spaghetti code. Because of this undocumented complex inheritance it took many tries to set up a hierarchy menu class set up with the help of many experts in the area of C++. The *pdmenu.h* and *pdmenu.c* was rewritten in order to get the hierarchy menus set up. In order to have a hierarchy of pull down menus the following classes were used:

1. Class Window. This is the window class. While there are many classes that the Window class has inherited, the lower classes does not need to be upgraded for our usage.

2. Class Interactor. Interactor class is the one that handles all the handshaking activities between all the modules. These activities are:

- Configuration.
- Traversing hierarchy.
- Input events.
- Outputs.
- Subject-view communications.
- Canvas properties.
- Top-level interactors.

3. Class InteractorWindow. An InteractorWindow will bind, unbind, receive, set attributes, and handle targets.

4. Class Scene. A Scene can hold many elements with functionality of Insert, Change, Remove, Move, Raise, Lower, and Propagate.

5. Class MonoScene. A MonoScene can hold only one element.

6. Class HighlighterParent. A HighlighterParent creates a highlight painter for its interior Highlighters to share.

7. Class Highlighter. A Highlighter draws itself and highlights or unhighlights itself on command.

8. Class PullDownMenuActivator. A PullDownMenuActivator displays a test label and opens a menu when it is activated. PullDownMenuActivator stores the bar it belongs to and its text label. It starts off with an empty menu and no stored commands, although it allocates some initial space to store the commands. It must catch the same mouse button PullDownMenuCommand catches.

9. Class PullDownMenuCommand. A PullDownMenuCommand displays a text label and executes a command. PullDownMenuCommand stores the activator it belongs to and its text labels. It catches only one mouse button to prevent the user from accidentally executing a command upon another button's release.

10. Class PullDownMenuDivider. A PullDownMenuDivider displays a horizontal line extending the full width of the menu, dividing it into two submenus.

11. Class PullDownMenuBar. A PullDownMenuBar displays several activators and coordinates which activator will open its menu. PullDownMenuBar starts with no currently active or stored activators although it allocates some initial space to store them.

12. Class PullDownMenuButton. A PullDownMenuButton displays a text label and executes a command.

Third problem was encountered during the connectivity between CAPS92 tools and the new editors' menu commands. CAPS92 did not use tools through pull down or push button menus. Instead the same tools' procedures that came with Idraw was copied, modified, and used.

Final problem was encountered during the conversion of input and output of the graphics editor. Because the initial version of the graphics editor was created before there was a syntax-directed editor for PSDL, it was designed in isolation. The CAPS92 version of editor.c uses two files to save and load the graphs. A PostScript file and a graph file. The PostScript file contains all of the information to make the graph appear on the screen, i.e.

all of the drawing information including fonts, colors, brushes, etc. The graph file tells the MET's of operators, and which edges hook to which operators. The identification numbers of the operators and edges actually come from the PostScript file. So, there was a great need to come up with a way of storing all the graph information needed to reconstruct the graph including fonts, colors, and all the other graphics information. The PostScript output file of the graph still needed in order to print hard copies of the prototypes' graph. So the final challenge was to rewrite the input and output procedures to be able to write all the prototype graphs' information, (Operators, Streams, METs, Latencies, and other needed information) in a file that be usable by SDE. Also the graphics editor must be able to read and display this file after modification by SDE.

C. GUIDANCE FOR FUTURE UPGRADES

It is important for anyone with a good knowledge of C++ who wants to upgrade anything in this graphics editor to take notes of the files which must get modified for addition or modification of one function. For example, in order to add the functionality for the Trigger_If button for constraints part of PSDL grammar, the following files had to be upgraded:

- dfd_defs.h
- dfdclasses.h
- drawing.h
- drawing.c
- editor.h
- editor.c
- keystrokes.h
- opsellist.h
- opsellist.c
- sloperator.h
- sloperator.c

- tools.c
- commands.c

D. NEEDED ADDITIONS FOR COMPLITION OF THE EDITOR

The graphics editor was designed to be user friendly and perform as many functions as needed to draw a prototype. Because of time limitations all the needed functionality of the command buttons were not implemented. The following functionalities are complete:

- Operators
- Streams (data streams, sampled data streams, states streams)
- Operators' label
- Steams' label
- Operator's MET
- Streams' latency
- Operators' triggered if condition
- Operators' decomposition
- Operators' atomic implementation in Ada
- Prototype's title/comment

Other needed procedures must be added in order to satisfy the execution of other command buttons. The existing codes from the implemented command buttons may be copied and changed to add the other desired functionalities.

V. USERS MANUAL

A. INTRODUCTION

The Computer Aided Prototyping System, CAPS, is a software development environment that provides a means to rapidly construct an executable prototype representing a large real-time software system on a software with real-time constraints.

A window based menu user interface guides the user through the rapid prototyping process. The interface can run on any UNIX workstation that uses the X windowing system.

B. USING THE GRAPHICS EDITOR

The graphics editor provides drawing tools and editing commands in order to create or modify an enhanced DFD. The tools choices are: Select, Move, Implement/Decompose, Title/Comment, Stream (represented as a line), Operator (represented as a circle).

The tools are used by placing the mouse pointer over the tool button and clicking with the left mouse button. To use the PSDL commands to input the prototypes' information, the Property hierarchy menu button must be activated. In order to see the pull down menus under the Property button, one must place the mouse pointer over the Property button and click and hold the left mouse button. Then while the left mouse button is pushed down, by moving the mouse pointer on top of the existing option menu buttons, they will get activated. The three pull down menus under the Property button are, Constrained_By, Stream, and Operator_Spec. By moving the mouse pointer while pressing the left mouse button over each of these buttons another menu will be activated. If the left mouse button is released or if the mouse pointer goes out of the menu parameter, all the menus will go away. So, in order to activate a command, the left mouse button must be held down and drag the mouse pointer to the desired command and then release the left mouse button.

TOOLS

a. Select

This tool is used to mark as selected those objects drawn on the screen that will be modified by another tool or command. Select a single graphics object, or select more than one object by holding down the left mouse button and dragging the mouse cursor until each of desired objects are enclosed in the resulting rectangle.

b. Move

This tool is used to move an operator, text, or part of the data flow diagram. A data stream or state cannot be moved by themselves. If an operator moves, its associated objects will move with it. Select a single graphics object, and move the object by holding down the left mouse button and dragging the mouse cursor. If more than one object is to be moved, we must first use the Select tool to select the group of objects to be moved, and then use the Move tool to drag the group to the new location.

c. Implement/Decompose

This tool is used to decompose an operator into a lower level DFD. First, select an operator. The prototype will automatically be saved. A dialog box will pop up to determine type of decomposition desired. The following choices are available:

• Graphics Editor: Another graphics editor will appear and the lower level DFD may be drawn. There is no limitation on the level of decomposition. Decomposition may be continued as many levels as there is need for.

• Ada: Generate the PSDL declaration for an Ada implementation of the operator. The actual Ada code must be written by the user.

• Search: Generate the PSDL declaration for an Ada implementation of the operator and search the software base for a reusable component to match the PSDL specification of the operator.

d. Title/Comment

This tool is used to add comments anywhere on the drawing. It is used by placing the mouse cursor where the text is to be placed and clicking with the left mouse button will activate the keyboard for text entry. The comment is not used by SDE. It is used for explaining a prototype or giving a name to a prototypes graph for clarity.

e. Stream

This tool is used to draw a data stream or state variable. At least one end point of a data flow must be inside an operator(vertex). In order to have a stream from operator A to operator B, first must place the mouse cursor inside the operator that stream comes from and press the left mouse button one time only, then drag the mouse cursor without pressing on any of the mouse buttons inside Operator B that streams goes to and press the mouse's right button. The end points of the stream will automatically get adjusted at the intersection of the operator and the stream. Figure 5.1 shows Stream_A_B from Operator A to Operator B.

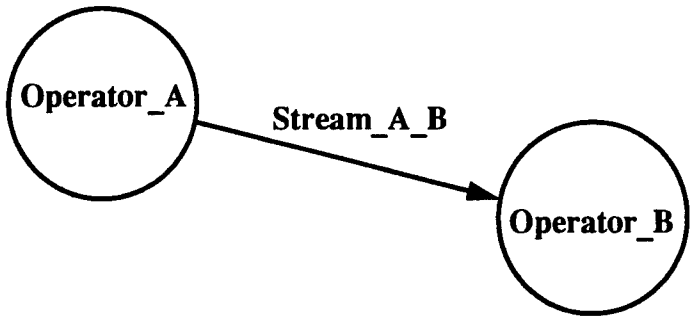

Figure 5.1: Stream_A_B from Operator A to B

In order to draw a spline between two Operators A and B, first place the mouse inside Operator A and press the left mouse button one time, next move the mouse cursor (without pressing any of the mouse buttons) to an intermediate point between A and B. press the left mouse button once. This will cause the spline to pass through this point. You may add as many intermediate points as required. Finally, move the mouse to Operator B and press the right mouse button. The spline will then be drawn from Operator A to Operator B, passing through all intermediate points. Figure 5.2 shows Stream_A_B from Operator A to Operator B through two intermediate points.

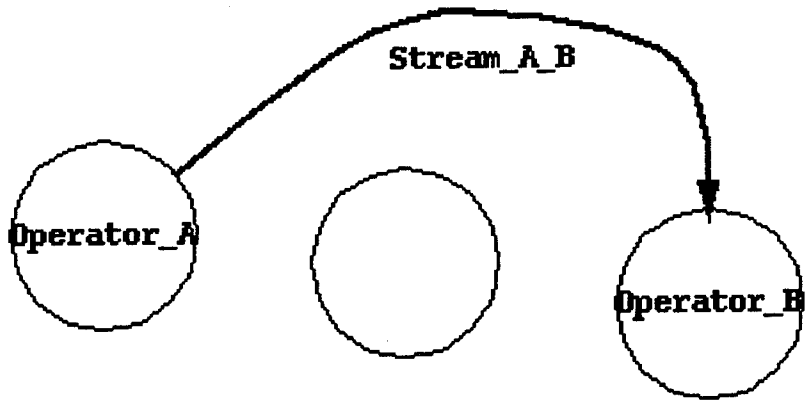

Figure 5.2: Stream_A_B from Operator A to Operator B

f. Operator

This tool is used to draw an operator. The radius of the operator will be 35 pixels. Choose were the center of the operator is desired by the mouse cursor and push the left mouse button to get an operator at that location.

COMMANDS

a. Property

Except the Operator and Stream that was put on the left panel as a tool, rest of the PSDL grammar has been implemented in the pull down menu bar hierarchy under the Property button.

b. Prototype

This pull down menu present prototype operations. The following operations are available under this menu:

- OPEN will open up a window with the list of existing prototypes.
- COMMIT
- PRINT will print the prototype's data flow diagram.
- QUIT will save the active prototype and quits the graphics editor.

c. Edit

This pull down menu presents editing operations. The following edit operations are available:

• DELETE will delete the selected object or objects.

• SELECT ALL will select all of the existing objects in the active prototype.

d. Graphics

This pull down menu presents the user with the following capabilities:

• Font, This pull down menu will present the user with list of fonts to be used for the text in the drawing. Only the selected text will use the chosen font.

• FgColor, this pull down menu give the user different options of color for the selected text, streams, and operators.

• BgColor, This pull down menu presents a list of background colors for the selected operators.

e. Option

This pull down menu presents various options to aid in putting the drawing on the page, such as reducing the prototypes' graph to fit on one page, centering the prototypes' graph on the page, and providing a grid with different settings.

VI. CONCLUSIONS AND RECOMMENDATIONS

A. SUMMARY

The goal of this thesis is to develope a friendly editing environment for computer aided rapid prototyping of hard real time systems for CAPS. This graphics editor constructs a graphical representation of the prototype.

The graphics editor handles all the editing requirements of a prototype. It provides many user friendly features, such as moving objects associated with an operator when the operator is moved and updating the PSDLs' associated parts, also updating the PSDL program associated with an operator when one of its associated objects is modified or deleted. The graphics editor also automatically generates a PSDL representation of the prototype.

Additional options were added to the original design as a result of user testing. The Title/Comment tool will give the users the capability to add explanation for any operator or stream. Another use of this tool is to add the title on the top of each prototype graph, so when someone look at the graph instantly understand what the prototype is all about, also this is useful for the decomposed operators, if an operator have three or four level of decomposition, the title on each graph may include an explanation of this condition. Another useful addition is that each decomposed operator will be marked with an asterisk on the top of its circle. When the Ada is chosen for Decompose/Implement option, an Emacs editor will be open with all the existing information implemented from the graph. In the editor there will be the package specifications and the package body with all the existing information from the graph. The unknown information will be shown as undefined. Professor Berzins recommended that we should have different lines thickness or types for streams (edges) to graphically be able to identify the difference between data streams, sampled streams and states. This accomplished, now the data streams are shown with heavy weight line, states with thin lines, and the sampled streams with heavy dashed lines.

There was also a suggestion on the color of the operators to be chosen automatically. After collecting ideas from the current users of CAPS, deciding on the different colors for certain types of operators, and thinking about the implementation part, concluded that:

- Each user will have their opinion on colors.
- Having predefined colors, and giving another option to the user to be able to over write the default colors need additional coding time that is not possible for this thesis time limits.
- In case of prototypes like Patriot and the Scud Missile, the predefined color will not look good for two separate systems.
- The users suggested that maybe the option of having no color is good when there is a need to print the prototypes graph for distribution.

B. CONCLUSION

As the result of this thesis, CAPS will have a powerful and user friendly graphics editor to rapidly construct a prototype of a large real-time system. The modification and changes made to the CAPS graphics editor as a result of this thesis, have made CAPS into a truly usable tool. SDE can now take advantage of this graphics editor to enhance the prototyping.

APPENDIX A: GRAPHICS EDITOR

A. STATEMENT OF PURPOSE

The purpose of the CAPS graphic editor is to establish a software development environment that will provide the designer with a means of constructing a prototype in an easy-to-use environment to facilitate the creation, modification, and maintenance of prototype.

B. GRAPHICS EDITER'S OUTPUT FORMAT

The following attempts to list the format of a .graph file for use with the latest version of the CAPS editor. This format is used to allow the graph to be saved and reconstructed from a single file, and eliminates the need for a.ps (PostScript) file to reconstruct the graph.

The files is composed of two sections, the first is a list of Vertices and Edges. The list may intermix Edge specifications with Vertex specifications with the following restriction. An Edge must be specified AFTER the Vertex/Vertices to which it is attached. The second portion of the file is simple the state of the editor at the point at which it the.graph file was last saved. The individual specifications are as follows:

(NOTE: the C++ style comments are for clarification only and do not appear in the actual.graph file)

```
// Vertex Specification:
2050        // The integer 2050 signifies an OPERATOR
int         // a unique OPERATOR identification number
int int     // CenterX, CenterY for the ellipse
int int     // RadiusX, RadiusY for ellipse
string int int int   // name and RGB values for Foreground
string int int int   // name and RGB values for Background
```

LABEL_STRING // see format for a label below

MET_STRING // see format for a label below

// Edge Specification

2052 // The integer 2052 signifies an Edge

int int int boolean // Number of points in the Spline

// Originating Operator id number

// Destination Operator id number

// Stream indicator 1 = stream, 0 = state

int int // a list of X, Y pairs for each point in Spline

string int int int // name and RGB values for Spline color

LABEL_STRING // see format for label below

LATENCY_STRING // This field should be present for Streams only

// it should not be present for states (SelfLoops)

// Strings are specified as follows:

int// length of the string excluding null terminator

// the value that would be returned by strlen()

char[]// the actual characters making up the string

// this may contain any character but \0

char[]// X name of print font defaults to

// "*-courier-medium-r-*-100-*

char[]// font family name default "Courier"

int // font size default 10

string int int int// name and RGB values for text color

boolean// default position indicator 1 = default

int int // Present only if default

// position field is 0

// text position x and y

// NOTE: if the length of a string is 0, i.e. no string has been

// specified for a particular label, then all other fields

// will be absent from the String specification

EXAMPLE:

An operator which has a label but no MET would look like this:

2050

1

69 112

35 35

Black 0 0 0

White 65280 65280 65280

17

Operator

Number_1

-courier-medium-r--100-*

Courier

10

Black 0 0 0

1

0

NOTE: notice that the label string contains a newline character

which is included in the string length.

Following the list of Vertices and Edges, the editor state appears as:

2098// The integer 2098 marks the end of the

// Vertex and Edge list

int int int int// x, y of view origin width, height of view

string int int int// The editors current foreground color name and RGB

string int int int// The editors current background color name and RGB

char[]// X name of print font defaults to

// "*-courier-medium-r-*-100-*

char[]// font family name default "Courier"

int // font size default 10

Final Note: There should be no blank lines in the.graph file blank lines shown above are only for clarity. The editor expects each line to be formatted properly with no more or less information on each line.

.graph file Grammar:

---

graph: component_s

END_MARKER NEW_LINE

editor_state

;

component_s: component

component_s

;

component: vertex | edge | comment

;

vertex: OPERATOR NEW_LINE

op_id

ellipse

fg_color

bg_color

label_text

met_text

;

edge: EDGE NEW_LINE

num_points from_op to_op is_stream NEW_LINE

point_s

fg_color

label_text

latency_text

;

comment: COMMENT NEW_LINE

comment_text

;

op_id:int NEW_LINE

;

ellipse:centerx centery NEW_LINE

radiusx radiusy NEW_LINE

;

centerx: int

;

centery: int

;

radiusx: int

;

radiusy: int

;

fg_color: color

```
;
bg_color: color
;
color: name red green blue NEW_LINE
;
name:string
;
red:int
;
green:int
;
blue:int
;
num_points: int
;
from_index:int
;
to_index:int
;
is_stream:boolean
;
point_s: point point_s
;
point: x y NEW_LINE
;
x: int
;
y:int
```

```
;
label_text:text
;
met_text:text
;
latency_text:text
;
comment_text:text
;
text: length
text_body
;
text_body: NULL
|
textstring
font
fg_color
position
;
position: default
| custom
  point
;
default: 0 NEW_LINE
;
custom: 1 NEW_LINE
;
length: int NEW_LINE
```

```
;
textstring: string NEW_LINE
;
font:printname
printfont
printsize
;
printname: string NEW_LINE
;
printfont: string NEW_LINE
;
printsize: int NEW_LINE
;
def_position: boolean NEW_LINE
;
editor_state:view_dimensions
fg_color
bg_color
font
;
view_dimensions: x y dimensions
;
dimensions: width height NEW_LINE
;
width: int
;
height: int
;
```

```
int:[0-9]+
;
boolean:1 | 0
;
string:[^\0 ]
;
NEW_LINE: '\n'
;
OPERATOR: 2050
;
EDGE:2052
;
COMMENT:2057
;
END_MARKER2098
;
```

APPENDIX B: PSDL GRAMMER

: The PSDL grammer is provided by Professor Luqi, Naval Postgraduate School in Monterey, California, and modified by Professor Valdis Berzins, Naval Postgraduate School in Monterey, California.
7/28/94: Valdis Berzins
Change op_id to allow adt operations to appear in the graph.

Affected productions:
op_id = id ["(" [id_list] "|" [id_list] ")"]

Restrict id's representing ada operator names to include an integer suffix representing a unique id. Restrict psdl operator names to include two integer suffixes, the first representing a unique id and the second representing an instance number. The instance number is zero for all operator declarations, and is nonzero for graph vertices, to represent operations of a psdl type that appear more than once in the expanded graph.

New productions:
op_name = ada_op_name "_" integer_literal
ada_op_name = id "_" integer_literal

Affected productions:
type_spec = "specification" ["generic" type_decl] [type_decl]
{"operator" id operator_spec} [functionality] "end"
operator = "operator" id operator_spec operator_impl
type_impl = "implementation" type_name {"operator" id operator_impl} "end"
operator_impl = "implementation ada" id "end"
initial_expression = type_name "." id ["(" initial_expression_list ")"]
expression = type_name "." id ["(" expression_list ")"]
op_id = id ["(" [id_list] "|" [id_list] ")"]
11/14/91: Change entered by Charlie Altizer.
The phrase "BY REQUIREMENTS" was changed to "REQUIRED BY."

Affected productions:
reqmts_trace = "by requirements" id_list
end Header

psdl grammar 12/1/90

Optional items are enclosed in [square brackets]. Items which may appear zero or more times appear in { braces }. Terminal symbols appear in " double quotes ". Groupings appear in (parentheses).

```
**************************************************************************

psdl
      = {component}

component
      = data_type
      | operator

data_type
      = "type" id type_spec type_impl

type_spec
      = "specification" ["generic" type_decl] [type_decl]
        {"operator" op_name operator_spec}
        [functionality] "end"

operator
      = "operator" op_name operator_spec operator_impl

operator_spec
      = "specification" {interface} [functionality] "end"

interface
      = attribute [reqmts_trace]

attribute
      = "generic" type_decl
      | "input" type_decl
      | "output" type_decl
      | "states" type_decl "initially" initial_expression_list
      | "exceptions" id_list
      | "maximum execution time" time

type_decl
      = id_list ":" type_name {"," id_list ":" type_name}

type_name
      = id
      | id "[" type_decl "]"

id_list
      = id {"," id}
```

```
reqmts_trace
    = "required by" id_list

functionality
    = [keywords] [informal_desc] [formal_desc]

keywords
    = "keywords" id_list

informal_desc
    = "description" "{" text "}"

formal_desc
    = "axioms" "{" text "}"

type_impl
    = "implementation ada" id "end"
    | "implementation" type_name {"operator" op_name operator_impl} "end"

operator_impl
    = "implementation ada" ada_op_name "end"
    | "implementation" psdl_impl "end"

psdl_impl
    = data_flow_diagram [streams] [timers] [control_constraints]
      [informal_desc]

data_flow_diagram
    = "graph" {vertex} {edge}

vertex
    = "vertex" op_id [":" time]
      -- time is the maximum execution time

edge
    = "edge" id [":" time] op_id "->" op_id
      -- time is the latency

op_id
    = [id "."] op_name ["(" [id_list] "|" [id_list] ")"]

streams
```

= "data stream" type_decl

timers
= "timer" id_list

control_constraints
= "control constraints" constraint {constraint}

constraint
= "operator" op_id
["triggered" [trigger] ["if" expression] [reqmts_trace]]
["period" time [reqmts_trace]]
["finish within" time [reqmts_trace]]
["minimum calling period" time [reqmts_trace]]
["maximum response time" time [reqmts_trace]]
{constraint_options}

constraint_options
= "output" id_list "if" expression [reqmts_trace]
| "exception" id ["if" expression] [reqmts_trace]
| timer_op id ["if" expression] [reqmts_trace]

trigger
= "by all" id_list
| "by some" id_list

timer_op
= "reset timer"
| "start timer"
| "stop timer"

initial_expression_list
= initial_expression {"," initial_expression}

initial_expression
= "true"
| "false"
| integer_literal
| real_literal
| string_literal
| id
| type_name "." op_name ["(" initial_expression_list ")"]
| "(" initial_expression ")"

```
    | initial_expression binary_op initial_expression
    | unary_op initial_expression

binary_op
    = "and" | "or" | "xor"
    | "<" | ">" | "=" | ">=" | "<=" | "/="
    | "+" | "-" | "&" | "*" | "/" | "mod" | "rem" | "**"
unary_op
    = "not" | "abs" | "-" | "+"

time
    = integer_literal unit

unit
    = "microsec"
    | "ms"
    | "sec"
    | "min"
    | "hours"

expression_list
    = expression {"," expression}

expression
    = "true"
    | "false"
    | integer_literal
    | time
    | real_literal
    | string_literal
    | id
    | type_name "." op_name ["(" expression_list ")"]
    | "(" expression ")"
    | initial_expression binary_op initial_expression
    | unary_op initial_expression

op_name
    = ada_op_name "_" integer_literal

ada_op_name
    = id "_" integer_literal

id
```

= letter {alpha_numeric}

real_literal

= integer_literal "." integer_literal

integer_literal

= digit {digit}

string_literal

= """" {char} """"

char

= any printable character except "}"

digit

= "0 .. 9"

letter

= "a .. z"

| "A .. Z"

| "_"

alpha_numberic

= letter

| digit

text

= {char}

APPENDIX C: CODES

Commands.h

```
#ifndef commands_h
#define commands_h
#include <InterViews/Std/stream.h>
#include "pdmenu.h"

// Declare imported types.

class Editor;
class MapKey;
class State;
// A Commands displays a PullDownMenuBar containing several
// PullDownMenuActivators each of which contains a PullDownMenu.

class Commands : public PullDownMenuBar {
public:
   Commands(Editor*, MapKey*, State*);
protected:
   void Init(Editor*, MapKey*, State*);
   void Reconfig();
};

#endif
```

Commands.c

```
*
* Changes made by : Mehdi E. Rowshanaee
#include "commands.h"
#include "editor.h"
#include "ipaint.h"
#include "istring.h"
#include "keystrokes.h"
#include "mapipaint.h"
#include "mapkey.h"
#include "sllines.h"
#include "state.h"
#include "pdmenu.h"
#include <InterViews/box.h>
#include <InterViews/painter.h>
#include <InterViews/sensor.h>
#include <InterViews/shape.h>

// An IdrawCommand enters itself into the MapKey so KeyEvents may be
// mapped to IdrawCommands.

class IdrawCommand : public PullDownMenuCommand {
public:
   IdrawCommand(PullDownMenuActivator*, const char*, char, Editor*,
                                                        MapKey* = nil);
protected:
   Editor* editor;                                      // handles drawing and editing operations
};
```

```
// IdrawCommand passes a printable string representing the given
// character for its key string and enters itself into the character's
// slot in the MapKey.

IdrawCommand::IdrawCommand (PullDownMenuActivator* a, const char* n, char c, Editor* e, MapKey* mk)
    : PullDownMenuCommand(a, n, mk ? mk->ToStr(c) : "")
  {
   editor = e;
   if (mk != nil)
   {
                    mk->Enter(this, c);
   }
}


// New additions to Commands.c  By: Mehdi Rowshanaee //
//    Sept 1994

class LabelCommand : public IdrawCommand {
public:
   LabelCommand (PullDownMenuActivator* a, Editor* e, MapKey* mk)
     : IdrawCommand(a, "Label", LABELCHAR, e, mk) { }
   void Execute (Event& e) {
    editor->ToolSet(LABELCHAR);
   }
};


class LatencyCommand : public IdrawCommand {
public:
   LatencyCommand (PullDownMenuActivator* a, Editor* e, MapKey* mk)
     : IdrawCommand(a, "Latency", LATENCYCHAR, e, mk) { }
   void Execute (Event& e) {
     editor->ToolSet(LATENCYCHAR);
   }
};

class Key_WordCommand : public IdrawCommand {
public:
   Key_WordCommand (PullDownMenuActivator* a, Editor* e, MapKey* mk)
     : IdrawCommand(a, "Key Word       ", KEYWORDCHAR, e, mk) { }
   void Execute (Event&) {
     // editor->HandleKey_Word();

   }
};



class  StatesCommand : public IdrawCommand {
public:
    StatesCommand (PullDownMenuActivator* a, Editor* e, MapKey* mk)
      : IdrawCommand(a, "States        ",STATESCHAR , e, mk) { }
    void Execute (Event& e ) {
      //  editor-> HandleStates(e);
   }
};


class  InformalCommand : public IdrawCommand {
public:
    InformalCommand (PullDownMenuActivator* a, Editor* e, MapKey* mk)
      : IdrawCommand(a, "Informal       ",INFORMALCHAR , e, mk) { }
    void Execute (Event&) {
      //  editor-> HandleInformal();
   }
};

class  FormalCommand : public IdrawCommand {
public:
    FormalCommand (PullDownMenuActivator* a, Editor* e, MapKey* mk)
      : IdrawCommand(a, "Formal         ",FORMALCHAR , e, mk) { }
    void Execute (Event&) {
      //  editor-> HandleFormal();
   }
```

```
};

class Exception_DeclCommand : public IdrawCommand {
public:
      Exception_DeclCommand (PullDownMenuActivator* a, Editor* e, MapKey* mk)
      : IdrawCommand(a, "Exception_Declaration",EXCEPTCHAR , e, mk) { }
     void Execute (Event&) {
      // editor->Exception_Decl();
   }
};

class Reset_TimerCommand : public IdrawCommand {
public:
      Reset_TimerCommand (PullDownMenuActivator* a, Editor* e, MapKey* mk)
      : IdrawCommand(a, "Reset Timer",RESET_TIMERCHAR , e, mk) { }
     void Execute (Event& e) {
      // editor->HandleReset_Timer(e);
   }
};

class Start_TimerCommand : public IdrawCommand {
public:
      Start_TimerCommand (PullDownMenuActivator* a, Editor* e, MapKey* mk)
      : IdrawCommand(a, "Start_Timer ",START_TIMERCHAR , e, mk) { }
     void Execute (Event& e) {
      // editor->HandleStart_Timer(e);
   }
};

class Stop_TimerCommand : public IdrawCommand {
public:
      Stop_TimerCommand (PullDownMenuActivator* a, Editor* e, MapKey* mk)
      : IdrawCommand(a, "Stop_Timer ",STOP_TIMERCHAR , e, mk) { }
     void Execute (Event& e) {
      // editor->HandleStop_Timer(e);
   }
};

class HourCommand : public IdrawCommand {
public:
      HourCommand (PullDownMenuActivator* a, Editor* e, MapKey* mk)
      : IdrawCommand(a, "Hour    ",HOURCHAR , e, mk) { }
     void Execute (Event& e) {
       editor->ToolSet(HOURCHAR);
   }
 };

 class LHourCommand : public IdrawCommand {
public:
      LHourCommand (PullDownMenuActivator* a, Editor* e, MapKey* mk)
      : IdrawCommand(a, "Hour    ",LHOURCHAR , e, mk) { }
     void Execute (Event& e) {
       editor->ToolSet(LHOURCHAR);
   }
 };


class Micro_SecCommand : public IdrawCommand {

public:
   Micro_SecCommand (PullDownMenuActivator* a, Editor* e, MapKey* mk)
   : IdrawCommand(a, "Micro Second", MICSECCHAR, e, mk) { }
   void Execute (Event& e) {
      editor->ToolSet(MICSECCHAR);
   }
 };

class LMicro_SecCommand : public IdrawCommand {

public:
   LMicro_SecCommand (PullDownMenuActivator* a, Editor* e, MapKey* mk)
   : IdrawCommand(a, "Micro Second", LMICSECCHAR, e, mk) { }
   void Execute (Event& e) {
      editor->ToolSet(LMICSECCHAR);
   }
```

```
};

class Mili_SecCommand : public IdrawCommand {

public:
   Mili_SecCommand (PullDownMenuActivator* a, Editor* e, MapKey* mk)
   : IdrawCommand(a, "MilliSecond", MILSECCHAR, e, mk) { }
   void Execute (Event& e) {
      editor->ToolSet(MILSECCHAR);
   }
};

class MinuteCommand : public IdrawCommand {

public:
   MinuteCommand (PullDownMenuActivator* a, Editor* e, MapKey* mk)
   : IdrawCommand(a, "Minute", MINUTECHAR, e, mk) { }
   void Execute (Event& e) {

      editor->ToolSet(MINUTECHAR);
   }
};

class SecondCommand : public IdrawCommand {

public:
   SecondCommand (PullDownMenuActivator* a, Editor* e, MapKey* mk)
   : IdrawCommand(a, "Second ", SECONDCHAR, e, mk) { }
   void Execute (Event& e) {

      editor->ToolSet(SECONDCHAR);

   }
};

class LMili_SecCommand : public IdrawCommand {

public:
   LMili_SecCommand (PullDownMenuActivator* a, Editor* e, MapKey* mk)
   : IdrawCommand(a, "MilliSecond", LMILSECCHAR, e, mk) { }
   void Execute (Event& e) {
      editor->ToolSet(LMILSECCHAR);
   }
};

class LMinuteCommand : public IdrawCommand {

public:
   LMinuteCommand (PullDownMenuActivator* a, Editor* e, MapKey* mk)
   : IdrawCommand(a, "Minute", LMINUTECHAR, e, mk) { }
   void Execute (Event& e) {

      editor->ToolSet(LMINUTECHAR);
   }
};

class LSecondCommand : public IdrawCommand {

public:
   LSecondCommand (PullDownMenuActivator* a, Editor* e, MapKey* mk)
   : IdrawCommand(a, "Second ", LSECONDCHAR, e, mk) { }
   void Execute (Event& e) {

      editor->ToolSet(LSECONDCHAR);

   }
};

class IfCommand : public IdrawCommand {

public:
   IfCommand (PullDownMenuActivator* a, Editor* e, MapKey* mk)
   : IdrawCommand(a, "Trigger IF", IFCHAR, e, mk) { }
   void Execute (Event& e) {
      editor->ToolSet(IFCHAR);
   }
```

```
};

class By_AllCommand : public IdrawCommand {

public:
  By_AllCommand (PullDownMenuActivator* a, Editor* e, MapKey* mk)
  : IdrawCommand(a, "Triggerd By All", BYALLCHAR, e, mk) { }
  void Execute (Event& e) {
    //      editor->HandleBy_All(e);
  }
};

class By_SomeCommand : public IdrawCommand {

public:
  By_SomeCommand (PullDownMenuActivator* a, Editor* e, MapKey* mk)
  : IdrawCommand(a, "Triggerd By Some", BYSOMECHAR, e, mk) { }
  void Execute (Event& e) {
    //      editor->HandleBy_Some(e);
  }
};


class IntegerCommand : public IdrawCommand {

public:
  IntegerCommand (PullDownMenuActivator* a, Editor* e, MapKey* mk)
  : IdrawCommand(a, "Integer", INTEGERCHAR, e, mk) { }
  void Execute (Event& e) {
    //      editor->HandleInteger(e);
  }
};

class RealCommand : public IdrawCommand {

public:
  RealCommand (PullDownMenuActivator* a, Editor* e, MapKey* mk)
  : IdrawCommand(a, "Real", REALCHAR, e, mk) { }
  void Execute (Event& e) {
    //      editor->HandleReal(e);
  }
};

class BooleanCommand : public IdrawCommand {

public:
  BooleanCommand (PullDownMenuActivator* a, Editor* e, MapKey* mk)
  : IdrawCommand(a, "Boolean", BOOLEANCHAR, e, mk) { }
  void Execute (Event& e) {
    //      editor->HandleBoolean(e);
  }
};

class User_DefinedCommand : public IdrawCommand {

public:
  User_DefinedCommand (PullDownMenuActivator* a, Editor* e, MapKey* mk)
  : IdrawCommand(a, "User Defined", USERDEFINEDCHAR, e, mk) { }
  void Execute (Event&e) {
    //      editor->HandleUser_Definede();
  }
};

class OpenCommand : public IdrawCommand {
public:
  OpenCommand (PullDownMenuActivator* a, Editor* e, MapKey* mk)
  : IdrawCommand(a, "Open...", OPENCHAR, e, mk) { }
  void Execute (Event&) {
                editor->Open();
  }
};

class CommitCommand : public IdrawCommand {
public:
  CommitCommand (PullDownMenuActivator* a, Editor* e, MapKey* mk)
  : IdrawCommand(a, "Commit", SAVECHAR, e, mk) { }
```

```
    void Execute (Event&) {
                editor->Commit();
    }
};

class PrintCommand : public IdrawCommand {
public:
    PrintCommand (PullDownMenuActivator* a, Editor* e, MapKey* mk)
    : IdrawCommand(a, "Print...", PRINTCHAR, e, mk) { }
    void Execute (Event&) {
                editor->Print();
    }
};

class QuitCommand : public IdrawCommand {
public:
    QuitCommand (PullDownMenuActivator* a, Editor* e, MapKey* mk)
    : IdrawCommand(a, "Quit", QUITCHAR, e, mk) { }
    void Execute (Event& e) {
                editor->Quit(e);
    }
};


class DeleteCommand : public IdrawCommand {
public:
    DeleteCommand (PullDownMenuActivator* a, Editor* e, MapKey* mk)
    : IdrawCommand(a, "Delete", DELETECHAR, e, mk) { }
    void Execute (Event&) {
                editor->Delete();
    }
};

class SelectAllCommand : public IdrawCommand {
public:
    SelectAllCommand (PullDownMenuActivator* a, Editor* e, MapKey* mk)
    : IdrawCommand(a, "Select All", SELECTALLCHAR, e, mk) { }
    void Execute (Event&) {
                editor->SelectAll();
    }
};

class FontCommand : public IdrawCommand {
public:
    FontCommand (PullDownMenuActivator* a, Editor* e, IFont* f)
    : IdrawCommand(a, f->GetPrintFontAndSize(), '\0', e) {
                font = f;
    }
    void Execute (Event&) {
                editor->SetFont(font);
    }
protected:
    void Reconfig () {
                Font* f = *font;
                if (output->GetFont() != f) {
                    Painter* copy = new Painter(output);
                    copy->Reference();
                    Unref(output);
                    output = copy;
                    output->SetFont(f);
                }
                IdrawCommand::Reconfig();
    }
    void Resize () {                                    // need constant left pad to line up entries
                const int xpad = 6;
                name_x = xpad;
                name_y = (ymax - output->GetFont()->Height() + 1) / 2;
                key_x = key_y = 0;
    }
    IFont* font;                                        // stores font to give Editor
};

static const int PICXMAX = 47;// chosen to minimize scaling for canvas
static const int PICYMAX = 14;
```

```
class PatternCommand : public IdrawCommand {
public:
  PatternCommand (PullDownMenuActivator* a, Editor* e, IPattern* p, State* s)
  : IdrawCommand(a, "None", '\0', e) {
                fgcolor = s->GetFgColor();
                bgcolor = s->GetBgColor();
                pattern = p;
                patindic = nil;
  }
  ~PatternCommand () {
                Unref(patindic);
  }
  void Execute (Event&) {
                editor->SetPattern(pattern);
  }
protected:
  void Reconfig () {
                IdrawCommand::Reconfig();
                if (patindic == nil) {
                  patindic = new Painter(output);
                  patindic->Reference();
                  patindic->SetColors(*fgcolor, *bgcolor);
                  patindic->SetPattern(*pattern);
                }
  }
  void Redraw (Coord l, Coord b, Coord r, Coord t) {
                if (pattern->None()) {
                  IdrawCommand::Redraw(l, b, r, t);
                } else {
                  output->ClearRect(canvas, l, b, r, t);
                  patindic->FillRect(canvas, name_x,name_y,xmax-name_x,ymax-name_y);
                  output->Rect(canvas, name_x, name_y, xmax-name_x, ymax-name_y);
                }
  }
  IColor* fgcolor;                                    // stores initial foreground color
  IColor* bgcolor;                                    // stores initial background color
  IPattern* pattern;                                  // stores pattern to give Editor
  Painter* patindic;                                  // fills rect to demonstrate pat's effect
};

class ColorCommand : public IdrawCommand {
public:
  ColorCommand (PullDownMenuActivator* a, Editor* e, IColor* c)
  : IdrawCommand(a, c->GetName(), '\0', e) {
                key = "  ";
                color = c;
                colorindic = nil;
  }
  ~ColorCommand () {
                key = nil;
                Unref(colorindic);
  }
protected:
  void Reconfig () {
                IdrawCommand::Reconfig();
                if (colorindic == nil) {
                  colorindic = new Painter(output);
                  colorindic->Reference();
                  colorindic->SetColors(*color, colorindic->GetBgColor());
                }
  }
  void Redraw (Coord l, Coord b, Coord r, Coord t) {
                IdrawCommand::Redraw(l, b, r, t);
                colorindic->FillRect(canvas, key_x, key_y, xmax-name_x, ymax-name_y);
                output->Rect(canvas, key_x, key_y, xmax-name_x, ymax-name_y);
  }
  IColor* color;                                      // stores color to give Editor
  Painter* colorindic;// fills rect to demonstrate color's effect
};

class FgColorCommand : public ColorCommand {
public:
  FgColorCommand (PullDownMenuActivator* a, Editor* e, IColor* c)
  : ColorCommand(a, e, c) { }
  void Execute (Event&) {
                editor->SetFgColor(color);
```

```
    }
};

class BgColorCommand : public ColorCommand {
public:
    BgColorCommand (PullDownMenuActivator* a, Editor* e, IColor* c)
    : ColorCommand(a, e, c) { }
    void Execute (Event&) {
                editor->SetBgColor(color);
    }
};


class ReduceCommand : public IdrawCommand {
public:
    ReduceCommand (PullDownMenuActivator* a, Editor* e, MapKey* mk)
    : IdrawCommand(a, "Reduce", REDUCECHAR, e, mk) { }
    void Execute (Event&) {
                editor->Reduce();
    }
};

class EnlargeCommand : public IdrawCommand {
public:
    EnlargeCommand (PullDownMenuActivator* a, Editor* e, MapKey* mk)
    : IdrawCommand(a, "Enlarge", ENLARGECHAR, e, mk) { }
    void Execute (Event&) {
                editor->Enlarge();
    }
};

class NormalSizeCommand : public IdrawCommand {
public:
    NormalSizeCommand (PullDownMenuActivator* a, Editor* e, MapKey* mk)
    : IdrawCommand(a, "Normal Size", NORMALSIZECHAR, e, mk) { }
    void Execute (Event&) {
                editor->NormalSize();
    }
};

class ReduceToFitCommand : public IdrawCommand {
public:
    ReduceToFitCommand (PullDownMenuActivator* a, Editor* e, MapKey* mk)
    : IdrawCommand(a, "Reduce To Fit", REDUCETOFITCHAR, e, mk) { }
    void Execute (Event&) {
                editor->ReduceToFit();
    }
};

class CenterPageCommand : public IdrawCommand {
public:
    CenterPageCommand (PullDownMenuActivator* a, Editor* e, MapKey* mk)
    : IdrawCommand(a, "Center Page", CENTERPAGECHAR, e, mk) { }
    void Execute (Event&) {
                editor->CenterPage();
    }
};

class RedrawPageCommand : public IdrawCommand {
public:
    RedrawPageCommand (PullDownMenuActivator* a, Editor* e, MapKey* mk)
    : IdrawCommand(a, "Redraw Page", REDRAWPAGECHAR, e, mk) { }
    void Execute (Event&) {
                editor->RedrawPage();
    }
};

class GriddingOnOffCommand : public IdrawCommand {
public:
    GriddingOnOffCommand (PullDownMenuActivator* a, Editor* e, MapKey* mk)
    : IdrawCommand(a, "Gridding on/off", GRIDDINGONOFFCHAR, e, mk) { }
    void Execute (Event&) {
                editor->GriddingOnOff();
    }
};
```

```
class GridVisibleInvisibleCommand : public IdrawCommand {
public:
  GridVisibleInvisibleCommand (PullDownMenuActivator* a,Editor* e,MapKey* mk)
  : IdrawCommand(a, "Grid visible/invisible", GRIDVISIBLEINVISIBLECHAR, e, mk) { }
  void Execute (Event&) {
                editor->GridVisibleInvisible();
  }
};

class GridSpacingCommand : public IdrawCommand {
public:
  GridSpacingCommand (PullDownMenuActivator* a, Editor* e, MapKey* mk)
  : IdrawCommand(a, "Grid spacing...", GRIDSPACINGCHAR, e, mk) { }
  void Execute (Event&) {
                editor->GridSpacing();
  }
};

class OrientationCommand : public IdrawCommand {
public:
  OrientationCommand (PullDownMenuActivator* a, Editor* e, MapKey* mk)
  : IdrawCommand(a, "Orientation", ORIENTATIONCHAR, e, mk) { }
  void Execute (Event&) {
                editor->Orientation();
  }
};

class ShowVersionCommand : public IdrawCommand {
public:
  ShowVersionCommand (PullDownMenuActivator* a, Editor* e, MapKey* mk)
  : IdrawCommand(a, "", SHOWVERSIONCHAR, e, mk) {
                Listen(noEvents);
  }
  void Execute (Event&) {
                editor->ShowVersion();
  }
protected:
  void Reconfig () {
                shape->width = shape->height = 0;
  }
};

// Commands creates its commands.

Commands::Commands (Editor* e, MapKey* mk, State* s) {

  Init(e, mk, s);
}

// Init creates the activators and commands, inserts the commands into
// menus, gives the menus to the activators, and inserts the activators.

void Commands::Init (Editor* e, MapKey* mk, State* state)
 {

  PullDownMenuActivator*  property= new PullDownMenuActivator(this, "Property" );
  PullDownMenuActivator*  prot    = new PullDownMenuActivator(this, "Prototype");
  PullDownMenuActivator*  edit    = new PullDownMenuActivator(this, "Edit"     );
  PullDownMenuActivator*  graphics   = new PullDownMenuActivator(this, "Graphics");
//  PullDownMenuActivator*  pat     = new PullDownMenuActivator(this, "Pattern" );
  PullDownMenuActivator*  option  = new PullDownMenuActivator(this, "Option"   );


 Scene* propertymenu = new VBox;

 PullDownMenuActivator* constraint_by =  new PullDownMenuActivator ( property , "Constraint_By ");
 PullDownMenuActivator* operator_spec =  new PullDownMenuActivator ( property , "Operator Specs");
 PullDownMenuActivator* stream        =  new PullDownMenuActivator ( property , "Stream       ");

 propertymenu -> Insert (constraint_by);
// propertymenu  -> Insert (new PullDownMenuDivider);
 propertymenu -> Insert (stream);
// propertymenu  -> Insert (new PullDownMenuDivider);
 propertymenu -> Insert (operator_spec);
```

```
Scene *op_spec_menu = new VBox;

PullDownMenuActivator *op_spec_informal = new PullDownMenuActivator (operator_spec , "Informal");
PullDownMenuActivator *op_spec_formal  = new PullDownMenuActivator (operator_spec , "Formal");
PullDownMenuActivator *op_spec_generic = new PullDownMenuActivator (operator_spec , "Generic    ");
PullDownMenuActivator *op_spec_input   = new PullDownMenuActivator (operator_spec , "Input      ");
PullDownMenuActivator *op_spec_output  = new PullDownMenuActivator (operator_spec , "Output     ");
PullDownMenuActivator *op_spec_met     = new PullDownMenuActivator (operator_spec , "M E T      ");
PullDownMenuActivator *latency         = new PullDownMenuActivator (stream       , "Latency    ");


op_spec_menu  -> Insert (new Key_WordCommand (operator_spec, e, mk ));
op_spec_menu  -> Insert (new StatesCommand ( operator_spec, e, mk ));
op_spec_menu  -> Insert (new InformalCommand( operator_spec, e, mk) );
op_spec_menu  -> Insert (new FormalCommand ( operator_spec, e, mk) );
op_spec_menu  -> Insert (op_spec_generic);
op_spec_menu  -> Insert (op_spec_input);
op_spec_menu  -> Insert (op_spec_output);
op_spec_menu  -> Insert (new Exception_DeclCommand ( operator_spec, e, mk ));
op_spec_menu  -> Insert (op_spec_met);

operator_spec -> SetMenu(op_spec_menu);

Scene *const_by_menu = new VBox;

PullDownMenuActivator *const_by_trigger = new PullDownMenuActivator (constraint_by, "Trigger    ");
PullDownMenuActivator *const_by_period  = new PullDownMenuActivator (constraint_by, "Period     ");
PullDownMenuActivator *const_by_finish  = new PullDownMenuActivator (constraint_by, "Finish within");
PullDownMenuActivator *const_by_min     = new PullDownMenuActivator (constraint_by, "Min Call   ");
PullDownMenuActivator *const_by_max     = new PullDownMenuActivator (constraint_by, "Max Response Time");
PullDownMenuActivator *const_by_timer   = new PullDownMenuActivator (constraint_by, "Timer      ");

const_by_menu -> Insert ( const_by_trigger );
const_by_menu -> Insert ( new PullDownMenuCommand (constraint_by, "Output Guard", "G") );
const_by_menu -> Insert ( const_by_finish );
const_by_menu -> Insert ( const_by_min );
const_by_menu -> Insert ( const_by_max );
const_by_menu -> Insert ( const_by_period );
const_by_menu -> Insert ( new PullDownMenuCommand (constraint_by, "Exception Condition", "X") );
const_by_menu -> Insert ( const_by_timer );
constraint_by -> SetMenu ( const_by_menu );


Scene *stream_menu = new VBox;
stream_menu  -> Insert (new LabelCommand (stream, e, mk ));
//stream_menu  -> Insert (new PullDownMenuDivider);
stream_menu  -> Insert (latency );
stream -> SetMenu (stream_menu);

Scene *latency_menu    = new VBox;
latency_menu -> Insert ( new LHourCommand     (latency , e, mk ));
// latency_menu -> Insert (new PullDownMenuDivider);
latency_menu -> Insert ( new LMinuteCommand   (latency , e, mk ));
// latency_menu -> Insert (new PullDownMenuDivider);
latency_menu -> Insert ( new LSecondCommand   (latency , e, mk ));
// latency_menu -> Insert (new PullDownMenuDivider);
latency_menu -> Insert ( new LMili_SecCommand (latency , e, mk ));
// latency_menu -> Insert (new PullDownMenuDivider);
latency_menu -> Insert ( new LMicro_SecCommand (latency , e, mk ));
latency -> SetMenu ( latency_menu);


Scene *trigger_menu = new VBox;
trigger_menu -> Insert ( new IfCommand     ( const_by_trigger, e, mk));
trigger_menu -> Insert ( new By_AllCommand  ( const_by_trigger, e, mk));
trigger_menu -> Insert ( new By_SomeCommand ( const_by_trigger, e, mk));
const_by_trigger -> SetMenu ( trigger_menu);

Scene *met_menu    = new VBox;
met_menu -> Insert ( new HourCommand     (op_spec_met , e, mk ));
// met_menu -> Insert (new PullDownMenuDivider);
met_menu -> Insert ( new MinuteCommand   (op_spec_met , e, mk ));
// met_menu -> Insert (new PullDownMenuDivider);
met_menu -> Insert ( new SecondCommand   (op_spec_met , e, mk ));
// met_menu -> Insert (new PullDownMenuDivider);
```

```
met_menu -> Insert ( new Mili_SecCommand  (op_spec_met , e, mk ));
// met_menu -> Insert (new PullDownMenuDivider);
 met_menu -> Insert ( new Micro_SecCommand (op_spec_met , e, mk ));
 op_spec_met -> SetMenu ( met_menu);

Scene *finish_menu  = new VBox;
finish_menu -> Insert ( new HourCommand     ( const_by_finish, e, mk));
//finish_menu -> Insert (new PullDownMenuDivider);
finish_menu -> Insert ( new MinuteCommand    ( const_by_finish, e, mk));
//finish_menu -> Insert (new PullDownMenuDivider);
finish_menu -> Insert ( new SecondCommand    ( const_by_finish, e, mk ));
//finish_menu -> Insert (new PullDownMenuDivider);
finish_menu -> Insert ( new Mili_SecCommand  ( const_by_finish, e, mk));
//finish_menu -> Insert (new PullDownMenuDivider);
finish_menu -> Insert ( new Micro_SecCommand ( const_by_finish, e, mk));
const_by_finish -> SetMenu ( finish_menu);

Scene *period_menu  = new VBox;
period_menu -> Insert ( new HourCommand     ( const_by_period, e, mk));
//period_menu -> Insert (new PullDownMenuDivider);
period_menu -> Insert ( new MinuteCommand    ( const_by_period, e, mk));
//period_menu -> Insert (new PullDownMenuDivider);
period_menu -> Insert ( new SecondCommand    ( const_by_period, e, mk ));
//period_menu -> Insert (new PullDownMenuDivider);
period_menu -> Insert ( new Mili_SecCommand  ( const_by_period, e, mk));
//period_menu -> Insert (new PullDownMenuDivider);
period_menu -> Insert ( new Micro_SecCommand ( const_by_period, e, mk));
const_by_period -> SetMenu ( period_menu);

Scene *min_menu    = new VBox;
min_menu   -> Insert ( new HourCommand     ( const_by_min, e, mk));
//min_menu   -> Insert (new PullDownMenuDivider);
min_menu   -> Insert ( new MinuteCommand    ( const_by_min, e, mk));
//min_menu   -> Insert (new PullDownMenuDivider);
min_menu   -> Insert (new SecondCommand     ( const_by_min, e, mk));
//min_menu   -> Insert (new PullDownMenuDivider);
min_menu   -> Insert ( new Mili_SecCommand  ( const_by_min, e, mk));
//min_menu   -> Insert (new PullDownMenuDivider);
min_menu   -> Insert ( new Micro_SecCommand ( const_by_min, e, mk));
const_by_min   -> SetMenu ( min_menu);

Scene *max_menu    = new VBox;
max_menu   -> Insert ( new HourCommand     ( const_by_max, e, mk));
//max_menu   -> Insert (new PullDownMenuDivider);
max_menu   -> Insert ( new MinuteCommand    ( const_by_max, e, mk));
//max_menu   -> Insert (new PullDownMenuDivider);
max_menu   -> Insert (new SecondCommand     (const_by_max, e, mk));
//max_menu   -> Insert (new PullDownMenuDivider);
max_menu   -> Insert ( new Mili_SecCommand  ( const_by_max, e, mk));
//max_menu   -> Insert (new PullDownMenuDivider);
max_menu   -> Insert ( new Micro_SecCommand ( const_by_max, e, mk));
const_by_max   -> SetMenu ( max_menu );


 Scene *generic_menu = new VBox;
 Scene *input_menu  = new VBox;
 Scene *output_menu  = new VBox;
 generic_menu -> Insert ( new IntegerCommand     ( op_spec_generic , e ,mk ));
// generic_menu -> Insert (new PullDownMenuDivider);
 generic_menu -> Insert ( new RealCommand        ( op_spec_generic , e ,mk ));
// generic_menu -> Insert (new PullDownMenuDivider);
 generic_menu -> Insert ( new BooleanCommand     ( op_spec_generic , e ,mk ));
// generic_menu -> Insert (new PullDownMenuDivider);
 generic_menu -> Insert ( new User_DefinedCommand ( op_spec_generic , e ,mk ));

 input_menu  -> Insert ( new IntegerCommand     ( op_spec_input  , e ,mk ));
// input_menu  -> Insert (new PullDownMenuDivider);
 input_menu  -> Insert ( new RealCommand        ( op_spec_input  , e ,mk ));
// input_menu  -> Insert (new PullDownMenuDivider);
 input_menu  -> Insert ( new BooleanCommand     ( op_spec_input  , e ,mk ));
// input_menu  -> Insert (new PullDownMenuDivider);
 input_menu  -> Insert ( new User_DefinedCommand ( op_spec_input  , e ,mk ));

 output_menu  -> Insert ( new IntegerCommand     ( op_spec_output , e ,mk ));
// output_menu  -> Insert (new PullDownMenuDivider);
 output_menu  -> Insert ( new RealCommand        ( op_spec_output , e ,mk ));
```

```
// output_menu -> Insert (new PullDownMenuDivider);
  output_menu -> Insert ( new BooleanCommand     ( op_spec_output , e ,mk ));
// output_menu -> Insert (new PullDownMenuDivider);
  output_menu -> Insert ( new User_DefinedCommand ( op_spec_output , e ,mk ));
  op_spec_generic -> SetMenu ( generic_menu);
  op_spec_input  -> SetMenu ( input_menu );
  op_spec_output -> SetMenu ( output_menu );



Scene * timer_menu = new VBox;
timer_menu -> Insert ( new Reset_TimerCommand  ( const_by_timer ,e ,mk ));
timer_menu -> Insert ( new Start_TimerCommand  ( const_by_timer ,e ,mk ));
timer_menu -> Insert ( new Stop_TimerCommand   ( const_by_timer ,e ,mk ));
const_by_timer -> SetMenu( timer_menu );

   Scene* protmenu = new VBox;
   protmenu ->Insert(new OpenCommand   (prot, e, mk));
   protmenu ->Insert(new CommitCommand (prot, e, mk));
   protmenu ->Insert(new PrintCommand  (prot, e, mk));
   protmenu ->Insert(new PullDownMenuDivider);
   protmenu ->Insert(new QuitCommand   (prot, e, mk));

   Scene* editmenu = new VBox;
   editmenu->Insert(new DeleteCommand(edit, e, mk));
   editmenu->Insert(new SelectAllCommand(edit, e, mk));

 Scene* graphicsmenu = new VBox;

 PullDownMenuActivator* font =  new PullDownMenuActivator ( graphics , "Font");
 PullDownMenuActivator* fgcolor =  new PullDownMenuActivator (graphics,"FgColor");
 PullDownMenuActivator* bgcolor =  new PullDownMenuActivator ( graphics , "BgColor");

   Scene* fontmenu = new VBox;
   MapIFont* mf = state->GetMapIFont();
   for (IFont* f = mf->First(); !mf->AtEnd(); f = mf->Next())
   {
                  fontmenu->Insert(new FontCommand(font, e, f));
   }

/*   Scene* patternmenu = new VBox;
   MapIPattern* mp = state->GetMapIPattern();
   for (IPattern* p = mp->First(); !mp->AtEnd(); p = mp->Next())
    {
                  patternmenu->Insert(new PatternCommand(pat, e, p, state));
   }*/

   Scene* fgcolormenu = new VBox;
   MapIColor* mfg = state->GetMapIFgColor();
   for (IColor* fg = mfg->First(); !mfg->AtEnd(); fg = mfg->Next())
    {
                  fgcolormenu->Insert(new FgColorCommand(fgcolor, e, fg));
   }

   Scene* bgcolormenu = new VBox;
   MapIColor* mbg = state->GetMapIBgColor();
   for (IColor* bg = mbg->First(); !mbg->AtEnd(); bg = mbg->Next())
   {
                  bgcolormenu->Insert(new BgColorCommand(bgcolor, e, bg));
   }

   font    -> SetMenu (fontmenu);
   fgcolor -> SetMenu (fgcolormenu);
   bgcolor -> SetMenu (bgcolormenu);

 graphicsmenu -> Insert (font);
 graphicsmenu -> Insert (fgcolor);
 graphicsmenu -> Insert (bgcolor);


   Scene* optionmenu = new VBox;
   optionmenu->Insert(new ReduceCommand          (option, e, mk));
   optionmenu->Insert(new EnlargeCommand         (option, e, mk));
   optionmenu->Insert(new NormalSizeCommand      (option, e, mk));
   optionmenu->Insert(new ReduceToFitCommand     (option, e, mk));
   optionmenu->Insert(new CenterPageCommand      (option, e, mk));
```

```
    optionmenu->Insert(new RedrawPageCommand        (option, e, mk));
    optionmenu->Insert(new PullDownMenuDivider);
    optionmenu->Insert(new GriddingOnOffCommand     (option, e, mk));
    optionmenu->Insert(new GridVisibleInvisibleCommand(option, e, mk));
    optionmenu->Insert(new GridSpacingCommand       (option, e, mk));
    optionmenu->Insert(new OrientationCommand       (option, e, mk));
    optionmenu->Insert(new ShowVersionCommand       (option, e, mk));


    property -> SetMenu (propertymenu);
    prot     -> SetMenu (protmenu);
    edit     -> SetMenu (editmenu);
//  pat      -> SetMenu (patternmenu);
graphics->SetMenu(graphicsmenu);
    option   -> SetMenu (optionmenu);


  Scene* activators = new HBox;

    activators->Insert(property);
    activators->Insert(prot);
    activators->Insert(edit);

activators->Insert(graphics);
//  activators->Insert(font);
//  activators->Insert(pat);
//  activators->Insert(fgcolor);
//  activators->Insert(bgcolor);
    activators->Insert(option);

    Insert(activators);
}

// Reconfig makes Commands' shape unstretchable but shrinkable.

void Commands::Reconfig () {
    PullDownMenuBar::Reconfig();
    shape->Rigid(hfil, 0, 0, 0);

};
```

Commentlist.h

```
#ifndef commentlist_h
#define commentlist_h

#include "list.h"

class Selection;
class TextSelection;

class CommentNode : public BaseNode {
public:
  CommentNode(TextSelection* ts) { txtsel = ts; }
  boolean SameValueAs(void* p) { return txtsel == p; }
  TextSelection* GetSelection() { return txtsel; }

protected:
```

```
 TextSelection* txtsel;     // points to an operator selection
};

class CommentList : public BaseList {
public:
 CommentList();
 void Append(CommentNode *);
 CommentNode* First();
 CommentNode* Last();
 CommentNode* Prev();
 CommentNode* Next();
 CommentNode* GetCur();
 CommentNode* Index(int);
 void Remove(TextSelection*);
 void SetCur(TextSelection*);
};

inline void CommentList::Append(CommentNode *cn) {
  BaseList::Append(cn);
}

inline CommentNode* CommentList::First() {
 return (CommentNode*) BaseList::First();
}

inline CommentNode* CommentList::Last() {
 return (CommentNode*) BaseList::Last();
}

inline CommentNode* CommentList::Prev() {
 return (CommentNode*) BaseList::Prev();
}

inline CommentNode* CommentList::Next() {
 return (CommentNode*) BaseList::Next();
}

inline CommentNode* CommentList::GetCur() {
 return (CommentNode*) BaseList::GetCur();
}

inline CommentNode* CommentList::Index(int index) {
 return (CommentNode*) BaseList::Index(index);
}

#end
```

Commentlist.c

```
#include "commentlist.h"
#include "selection.h"
#include "sltext.h"

CommentList::CommentList() : BaseList() {
}

// Remove an operator from the list of operator selections

void CommentList::Remove(TextSelection* ts)  {
  SetCur(ts);
  if (!AtEnd()) {
    DeleteCur();
    delete ts;
  }
}

void CommentList::SetCur(TextSelection *ts) {
  for (CommentNode *cn = First(); !AtEnd(); cn = Next())
    if (ts == cn->GetSelection()) break;
}
```

dfd_defs.h

```
#ifndef dfd_defs_h
#define dfd_defs_h

#include <InterViews/defs.h>
```

```
// pixel radius of operator (ellipse)
#define  OperatorRadius 35

// number of characters in PSDL representation of operator
#define TXTBUFLEN 5000

// max number of prototypes in prototype directory
#define MAXPROTOTYPES 100

// name of scratch file used to edit PSDL for operator
#define PSDL_FILE "psdl.scratch"

// maximum length of message for message block
#define MAXMSGLEN 150

// name of scratch file used to write PSDL streams
#define STREAMS_FILE "streams.scratch"

// name of scratch file used to write PSDL mets
#define MET_FILE "mets.scratch"

// name of scratch file used to write PSDL constraints
#define CONSTRAINTS_FILE "constraints.scratch"

// name of file extensions

#define GRAPH_EXT ".ps"
#define GRAPH_EXT_LEN 3
#define IMP_PSDL_EXT ".imp.psdl"
#define IMP_EXT_LEN 9
#define SPEC_PSDL_EXT ".spec.psdl"
#define SPEC_EXT_LEN 10
#define DFD_EXT ".graph"
#define DFD_EXT_LEN 6

// keywords to be inserted when creating PSDL
#define OPER_TKN      "OPERATOR "
#define SPEC_TKN      " SPECIFICATION\n"
#define INPUT_TKN     "  INPUT\n"
#define OUTPUT_TKN    "  OUTPUT\n"
#define ST_TKN        "  STATES\n  UNDEFINED_ID : UNDEFINED_TYPE\n  INITIALLY\n  UNDEFINED_EXPRESSION\n"
#define MET_TKN       "  MAXIMUM EXECUTION TIME "
#define DESC_TKN      "  DESCRIPTION "
#define TEXT_TKN      "{ UNDEFINED_TEXT }\n"
#define END_TKN       "END\n"
#define IMP_TKN       "IMPLEMENTATION\n"
#define GR_TKN        " GRAPH\n"
#define VER_TKN       "  VERTEX "
#define EDGE_TKN      "  EDGE "
#define ID_TKN        "UNDEFINED_ID"
#define EXT_TKN       "EXTERNAL"
#define TYPE_DECL_TKN "UNDEFINED_ID : UNDEFINED_TYPE"
#define IMP_ADA_TKN   " IMPLEMENTATION ADA "
#define STREAM_TKN    " DATA STREAM\n"
#define CON_TKN       " CONTROL CONSTRAINTS\n"
#define CON_OP_TKN    " OPERATOR UNDEFINED_ID\n"

#define TRIGGER_TKN       " TRIGGER CONDITION \n"

#define TRIGGER_IF_TKN    " TRIGGER IF\n "
#define TRIGGER_BY_ALL_TKN  " TRIGGER BY ALL\n"
#define TRIGGER_BY_SOME_TKN " TRIGGER BY SOME\n "
#define FORMAL_TKN        " FORMAL \n"
#define INFORMAL_TKN      " INFORMAL\n "
#define PERIOD_TKN        " PERIOD \n"
#define FINISH_WITHIN_TKN  " FINISH WITHIN \n"
#define MAX_RES_TIM_TKN   " MAXXIMUM RESPONSE TIME \n"
#define OUTPUT_GUARD_TKN   " OUTPUT GUARD\n "



// keywords to be used for search through PSDL text buffer.  They are
// different from those above because the text buffer can't ever locate
// newlines
#define INPUT_SCH_TKN  "INPUT"
#define SPEC_SCH_TKN   "SPECIFICATION"
```

```
#define OUTPUT_SCH_TKN "OUTPUT"
#define GEN_SCH_TKN    "GENERIC"
#define STATES_SCH_TKN "STATES"
#define EXCEPT_SCH_TKN "EXCEPTIONS"
#define MET_SCH_TKN    "MAXIMUM EXECUTION TIME "
#define MCP_SCH_TKN    "MINIMUM CALLING PERIOD"
#define MRT_SCH_TKN    "MAXIMUM RESPONSE TIME"
#define KEY_SCH_TKN    "KEYWORDS"
#define DESC_SCH_TKN   "DESCRIPTION"
#define AX_SCH_TKN     "AXIOM"
#define END_SCH_TKN    "END"
#define STREAM_SCH_TKN "DATA STREAM"
#define TIMER_SCH_TKN  "TIMER"
#define CON_SCH_TKN    "CONTROL CONSTRAINTS"

#define TRIGGER_SCH_TKN    " TRIGGER CONDITION "

#define FINISH_WITHIN_SCH_TKN " FINISH WITHIN "
#define TRIGGER_IF_SCH_TKN    " TRIGGER IF "
#define TRIGGER_BY_ALL_SCH_TKN" TRIGGER BY ALL "
#define TRIGGER_BY_SOME_SCH_TKN " TRIGGER BY SOME "
#define FORMAL_SCH_TKN         " FORMAL"
#define INFORMAL_SCH_TKN       " INFORMAL "
#define PERIOD_SCH_TKN         " PERIOD "
#define MAX_RES_TIM_SCCH_TKN   "  MAXXIMUM RESPONSE TIME "
#define OUTPUT_GUARD_SCH_TKN   " OUTPUT GUARD "


#define MET_SDE         "psdl_editor"
#define STREAMS_SDE     "psdl_editor"
#define CONSTRAINTS_SDE  "psdl_editor"
#define SPECIFICATION_SDE "psdl_editor"
#endif
```

dfd_classes.h

```
 * Changes made by : Mehdi Rowshanaee
* September 1994
*
*/

#ifndef dfdclasses_h
#define dfdclasses_h

static const int OPERATOR      = 2050;
static const int TERMINATOR    = 2051;
static const int DATAFLOW_SPLINE = 2052;
static const int SELFLOOP      = 2053;
static const int LABEL_OP      = 2054;
static const int LABEL_DF      = 2055;
static const int LABEL_SL      = 2056;
static const int COMMENT       = 2057;
static const int MET_OP        = 2058;
static const int LAT_DF        = 2059;
static const int END_MARKER    = 2098;
static const int NONE          = 2099;

static const int TRIG_IF_CONS   = 2101;
static const int TRIG_AL_CONS   = 2102;
static const int TRIG_SM_CONS   = 2103;

#endif
```

dialogbox.h

```
#ifndef dialogbox_h
#define dialogbox_h

#include <InterViews/filechooser.h>

// Declare imported types.

class ButtonState;
class IMessage;
```

```
class StringEditor;

// A DialogBox knows how to set its message and warning text and how
// to pop up itself over the underlying Interactor.

class DialogBox : public MonoScene {
public:

  void SetMessage(const char* = nil, const char* = nil);
  void SetWarning(const char* = nil, const char* = nil);
  void SetUnderlying(Interactor*);

protected:

  DialogBox(Interactor*, const char* = nil);

  void PopUp();
  void Disappear();

  IMessage* message;                                    // displays message text
  IMessage* warning;                                    // displays warning text
  Interactor* underlying;// we'll insert ourselves into its parent

};

// A Messager displays a message until it's acknowledged.

class Messager : public DialogBox {
public:

  Messager(Interactor*, const char* = nil);
  ~Messager();

  void Display();

protected:

  void Init();
  void Reconfig();

  ButtonState* ok;                                      // stores status of "ok" button
  Interactor* okbutton;// displays "ok" button

};

// A Confirmer displays a message until it's confirmed or cancelled.

class Confirmer : public DialogBox {
public:

  Confirmer(Interactor*, const char* = nil);
  ~Confirmer();

  char Confirm();

protected:

  void Init();
  void Reconfig();

  ButtonState* yes;                                     // stores status of "yes" button
  ButtonState* no;                                      // stores status of "no" button
  ButtonState* cancel;// stores status of "cancel" button
  Interactor* yesbutton;// displays "yes" button
  Interactor* nobutton;// displays "no" button
  Interactor* cancelbutton;// displays "cancel" button

};

// A Namer displays a string until it's edited or cancelled.

class Namer : public DialogBox {
public:

  Namer(Interactor*, const char* = nil);
  ~Namer();
```

```
    char* Edit(const char*);

protected:

    void Init();
    void Reconfig();

    ButtonState* accept;// stores status of "accept" button
    ButtonState* cancel;// stores status of "cancel" button
    Interactor* acceptbutton;// displays "accept" button
    Interactor* cancelbutton;// displays "cancel" button
    StringEditor* stringeditor;// displays and edits a string

};

// A Finder browses the file system and returns a file name.

class Finder : public FileChooser {
public:

    Finder(Interactor*, const char*);

    const char* Find();

protected:

    Interactor* Interior();
    boolean Popup(Event&, boolean = true);

protected:

    Interactor* underlying;// we'll insert ourselves into its parent

};

// A Chooser displays a set of choices and are displayed until one is
// chosen or is cancelled.

class Chooser : public DialogBox {
public:

    Chooser(Interactor*, const char*, const char*, const char*, const char*);
    ~Chooser();

    char Choose();

protected:

    void Init();
    void Reconfig();

    ButtonState* bs_1;                  // stores status of first button
    ButtonState* bs_2;                  // stores status of second button
                                        ButtonState* bs_3;   // stores status of third button

    ButtonState* cancel;// stores status of "cancel" button
    Interactor* button_1;// displays first button
    Interactor* button_2;// displays second button
                                        Interactor* button_3; // displays third button
    Interactor* cancelbutton;// displays "cancel" button

};
#endif
```

dialogbox.c

```
#include "dialogbox.h"
#include "istring.h"
#include <InterViews/box.h>
#include <InterViews/button.h>
#include <InterViews/canvas.h>
#include <InterViews/event.h>
#include <InterViews/font.h>
#include <InterViews/frame.h>
```

```
#include <InterViews/glue.h>
#include <InterViews/message.h>
#include <InterViews/painter.h>
#include <InterViews/sensor.h>
#include <InterViews/shape.h>
#include <InterViews/streditor.h>
#include <InterViews/world.h>

#include <InterViews/Std/os/fs.h>
#include <sys/param.h>

/*
 * An IMessage displays its own text, not somebody else's.
 */

class IMessage : public Message {
public:
    IMessage(const char* = nil, Alignment a = Center);
    ~IMessage();

    void SetText(const char* = nil, const char* = nil);
protected:
    char* buffer;                                   /* stores own copy of text */
};

/*
 * IMessage creates a buffer to store its own copy of the text.
 */

IMessage::IMessage (const char* msg, Alignment a) : Message(nil, a) {
    buffer = strdup(msg ? msg : "");
    text = buffer;
}

/*
 * Free storage allocated for the text buffer.
 */

IMessage::~IMessage () {
    delete buffer;
}

/*
 * SetText stores the new text and changes the IMessage's shape to fit
 * the new text's width.
 */

void IMessage::SetText (const char* beg, const char* end) {
    beg = beg ? beg : "";
    end = end ? end : "";
    delete buffer;
    buffer = new char[strlen(beg) + strlen(end) + 1];
    strcpy(buffer, beg);
    strcat(buffer, end);
    text = buffer;
    if (canvas != nil && canvas->Status() == CanvasMapped) {
                Reconfig();
                Parent()->Change(this);
    }
}

/*
 * DialogBox creates two IMessages to display a message and a warning
 * and stores its underlying Interactor.  DialogBox won't delete the
 * IMessages so its derived classes can put them in boxes which will
 * delete them when the boxes are deleted.
 */

DialogBox::DialogBox (Interactor* u, const char* msg) {
    SetCanvasType(CanvasSaveUnder); /* speed up expose redrawing if possible */
    input = allEvents;
    input->Reference();
    message = new IMessage(msg);
    warning = new IMessage;
    underlying = u;
}
```

```
/*
 * SetMessage sets the message's text.
 */

void DialogBox::SetMessage (const char* beg, const char* end) {
    message->SetText(beg, end);
}

/*
 * SetWarning sets the warning's text.
 */

void DialogBox::SetWarning (const char* beg, const char* end) {
    warning->SetText(beg, end);
}

/*
 * SetUnderlying sets the underlying Interactor over which the
 * DialogBox will pop up itself.
 */

void DialogBox::SetUnderlying (Interactor* u) {
    underlying = u;
}

/*
 * PopUp pops up the DialogBox centered over the underlying
 * Interactor's canvas.
 */

void DialogBox::PopUp () {
    World* world = underlying->GetWorld();
    Coord x, y;
    underlying->Align(Center, 0, 0, x, y);
    underlying->GetRelative(x, y, world);
    world->InsertTransient(this, underlying, x, y, Center);
}

/*
 * Disappear removes the DialogBox.  Since the user should see
 * warnings only once, Disappear clears the warning's text so the next
 * PopUp won't display it.
 */

void DialogBox::Disappear () {
    Parent()->Remove(this);
    SetWarning();
    Sync();
}

/*
 * Messager creates its button state and initializes its view.
 */

Messager::Messager (Interactor* u, const char* msg) : DialogBox(u, msg) {
    ok = new ButtonState(false);
    okbutton = new PushButton("  OK  ", ok, true);
    Init();
}

/*
 * Free storage allocated for the message's button state.
 */

Messager::~Messager () {
    Unref(ok);
}

/*
 * Display pops up the Messager and removes it when the user
 * acknowledges the message.
 */

void Messager::Display () {
    ok->SetValue(false);
```

```
    PopUp();

    int okay = false;
    while (!okay) {
                    Event e;
                    Read(e);
                    if (e.eventType == KeyEvent && e.len > 0) {
                      switch (e.keystring[0]) {
                      case '\r':                                    /* CR */
                      case '\007':                                  /* ^G */
                                                                    ok->SetValue(true);
                                                                    break;

                      default:
                                                                    break;

                      }
                    } else if (e.target == okbutton) {
                      e.target->Handle(e);
                    }
                    ok->GetValue(okay);
    }

    Disappear();
}

/*
 * Init composes Messager's view with boxes, glue, and frames.
 */

void Messager::Init () {
    SetClassName("Messager");

    VBox* vbox = new VBox;
    vbox->Align(Center);
    vbox->Insert(new VGlue);
    vbox->Insert(warning);
    vbox->Insert(new VGlue);
    vbox->Insert(message);
    vbox->Insert(new VGlue);
    vbox->Insert(okbutton);
    vbox->Insert(new VGlue);

    Insert(new Frame(vbox, 2));
}

/*
 * Reconfig pads Messager's shape to make the view look less crowded.
 */

void Messager::Reconfig () {
    DialogBox::Reconfig();
    Font* font = output->GetFont();
    shape->width += 2 * font->Width("mmmm");
    shape->height += 4 * font->Height();
}

/*
 * Confirmer creates its button states and initializes its view.
 */

Confirmer::Confirmer (Interactor* u, const char* prompt) : DialogBox(u, prompt) {
    yes                                                             = new ButtonState(false);
    no                                                              = new ButtonState(false);
    cancel              = new ButtonState(false);
    yesbutton    = new PushButton(" Yes  ", yes, true);
    nobutton     = new PushButton("  No  ", no, true);
    cancelbutton = new PushButton("Cancel", cancel, true);
    Init();
}

/*
 * Free storage allocated for the button states.
 */

Confirmer::~Confirmer () {
    Unref(yes);
```

```
    Unref(no);
    Unref(cancel);
}

/*
 * Confirm pops up the Confirmer, lets the user confirm the message or
 * not, removes the Confirmer, and returns the confirmation.
 */

char Confirmer::Confirm () {
    yes->SetValue(false);
    no->SetValue(false);
    cancel->SetValue(false);

    PopUp();

    int confirmed = false;
    int denied = false;
    int cancelled = false;
    while (!confirmed && !denied && !cancelled) {
                Event e;
                Read(e);
                if (e.eventType == KeyEvent && e.len > 0) {
                    switch (e.keystring[0]) {
                    case 'y':
                    case 'Y':
                                                        yes->SetValue(true);
                                                        break;

                    case 'n':
                    case 'N':
                                                        no->SetValue(true);
                                                        break;
                    case '\r':                          /* CR */
                    case '\007':                        /* ^G */
                                                        cancel->SetValue(true);
                                                        break;

                    default:
                                                        break;

                    }
                } else if (e.target == yesbutton || e.target == nobutton ||
                                                                e.target == cancelbutton)

                {
                    e.target->Handle(e);
                }
                yes->GetValue(confirmed);
                no->GetValue(denied);
                cancel->GetValue(cancelled);
    }

    Disappear();

    char answer = 'n';
    answer = confirmed ? 'y' : answer;
    answer = cancelled ? 'c' : answer;
    return answer;
}

/*
 * Init composes Confirmer's view with boxes, glue, and frames.
 */

void Confirmer::Init () {
    SetClassName("Confirmer");

    HBox* buttons = new HBox;
    buttons->Insert(new HGlue);
    buttons->Insert(yesbutton);
    buttons->Insert(new HGlue);
    buttons->Insert(nobutton);
    buttons->Insert(new HGlue);
    buttons->Insert(cancelbutton);
    buttons->Insert(new HGlue);

    VBox* vbox = new VBox;
    vbox->Align(Center);
    vbox->Insert(new VGlue);
```

```
    vbox->Insert(warning);
    vbox->Insert(new VGlue);
    vbox->Insert(message);
    vbox->Insert(new VGlue);
    vbox->Insert(buttons);
    vbox->Insert(new VGlue);

    Insert(new Frame(vbox, 2));
}

/*
 * Reconfig pads Confirmer's shape to make the view look less crowded.
 */

void Confirmer::Reconfig () {
    DialogBox::Reconfig();
    Font* font = output->GetFont();
    shape->width += 4 * font->Width("mmmm");
    shape->height += 4 * font->Height();
}

/*
 * Namer creates its button states and initializes its view.
 */

Namer::Namer (Interactor* u, const char* prompt) : DialogBox(u, prompt) {
    accept = new ButtonState(false);
    cancel = new ButtonState(false);
    acceptbutton = new PushButton(" OK ", accept, true);
    cancelbutton = new PushButton("Cancel", cancel, true);
    const char* sample = "                                        ";
    stringeditor = new StringEditor(accept, sample, "\007\015");
    stringeditor->Message("");
    Init();
}

/*
 * Free storage allocated for the button states.
 */

Namer::~Namer () {
    Unref(accept);
    Unref(cancel);
}

/*
 * Edit pops up the Namer, lets the user edit the given string,
 * removes the Namer, and returns the edited string unless the user
 * cancelled it.
 */

char* Namer::Edit (const char* string) {
    accept->SetValue(false);
    cancel->SetValue(false);
    if (string != nil) {
                stringeditor->Message(string);
    }
    stringeditor->Select(0, strlen(stringeditor->Text()));

    PopUp();

    int accepted = false;
    int cancelled = false;
    while (!accepted && !cancelled) {
                stringeditor->Edit();
                accept->GetValue(accepted);
                if (accepted == '\007') {
                    accept->SetValue(false);
                    cancel->SetValue(true);
                } else if (accepted == '\015') {
                    accept->SetValue(true);
                    cancel->SetValue(false);
                } else {
                    Event e;
                    Read(e);
                    if (e.target == acceptbutton || e.target == cancelbutton) {
```

```
                    e.target->Handle(e);
                }
            }
            accept->GetValue(accepted);
            cancel->GetValue(cancelled);
    }

    Disappear();

    char* result = nil;
    if (accepted) {
            const char* text = stringeditor->Text();
            if (text[0] != '\0') {
                result = strdup(text);
            }
    }
    return result;
}

/*
 * Init composes Namer's view with boxes, glue, and frames.
 */

void Namer::Init () {
    SetClassName("Namer");

    HBox* hboxedit = new HBox;
    hboxedit->Insert(new HGlue(5, 0, 0));
    hboxedit->Insert(stringeditor);
    hboxedit->Insert(new HGlue(5, 0, 0));

    VBox* vboxedit = new VBox;
    vboxedit->Insert(new VGlue(2, 0, 0));
    vboxedit->Insert(hboxedit);
    vboxedit->Insert(new VGlue(2, 0, 0));

    HBox* buttons = new HBox;
    buttons->Insert(new HGlue);
    buttons->Insert(acceptbutton);
    buttons->Insert(new HGlue);
    buttons->Insert(cancelbutton);
    buttons->Insert(new HGlue);

    VBox* vbox = new VBox;
    vbox->Align(Center);
    vbox->Insert(new VGlue);
    vbox->Insert(warning);
    vbox->Insert(new VGlue);
    vbox->Insert(message);
    vbox->Insert(new VGlue);
    vbox->Insert(new Frame(vboxedit, 1));
    vbox->Insert(new VGlue);
    vbox->Insert(buttons);
    vbox->Insert(new VGlue);
    vbox->Propagate(false);/* for reshaping stringeditor w/o looping */

    Insert(new Frame(vbox, 2));
}

/*
 * Reconfig pads Namer's shape to make the view look less crowded.
 */

void Namer::Reconfig () {
    DialogBox::Reconfig();
    Shape s = *stringeditor->GetShape();
    s.Rigid();
    stringeditor->Reshape(s);
    Font* font = output->GetFont();
    shape->width += 2 * font->Width("mmmm");
    shape->height += 4 * font->Height();
}

static const char* abspath (const char* file = nil) {
    const int bufsize = MAXPATHLEN+1;
    static char buf[bufsize];
```

```
    getcwd(buf, bufsize);
    strcat(buf, "/");

    if (file != nil) {
        strcat(buf, file);
    }
    return buf;
}

Finder::Finder (
    Interactor* u, const char* t
) : FileChooser(new ButtonState, abspath(), 10, 24, Center) {
    underlying = u;
    Init("", t);
    Insert(Interior());
}

static const char* ChdirIfNecessary (Finder* finder) {
    static char buf[MAXPATHLEN+1];
    const char* filename = finder->Choice();
    strcpy(buf, filename);
    char* bufptr = strrchr(buf, '/');

    if (bufptr != NULL) {
        *bufptr = '\0';
        if (chdir(buf) == 0) {
            filename = ++bufptr;
            finder->Message(abspath(filename));
        }
    }
    finder->SelectFile();
    return filename;
}

const char* Finder::Find () {
    const char* name = nil;
    Event e;
    if (Popup(e)) {
                name = ChdirIfNecessary(this);
    }
    return name;
}

Interactor* Finder::Interior () {
    return new Frame(FileChooser::Interior(" Open "), 2);
}

boolean Finder::Popup (Event&, boolean) {
    World* world = underlying->GetWorld();
    Coord x, y;
    underlying->Align(Center, 0, 0, x, y);
    underlying->GetRelative(x, y, world);
    world->InsertTransient(this, underlying, x, y, Center);
    boolean accepted = Accept();
    world->Remove(this);
    SetTitle("");
    return accepted;
}

/*
 * Chooser creates its button states and initializes its view.
 */

Chooser::Chooser (Interactor* u, const char* prompt, const char* label_1,
                                                    const char* label_2, const char* label_3) : DialogBox(u, prompt) {
    bs_1                                            = new ButtonState(false);
    bs_2                                            = new ButtonState(false);
                                                    bs_3    = new ButtonState(false);

    cancel              = new ButtonState(false);
    button_1    = new PushButton(label_1, bs_1, true);
    button_2    = new PushButton(label_2, bs_2, true);
                                                    button_3    = new PushButton(label_3, bs_3, true);
    cancelbutton = new PushButton("   Cancel   ", cancel, true);
    Init();
}
```

```
/*
 * Free storage allocated for the button states.
 */

Chooser::~Chooser () {
    Unref(bs_1);
    Unref(bs_2);
                                                        Unref(bs_3);

    Unref(cancel);
}

/*
 * Choose pops up the Chooser, lets the user choose one of the buttons,
 * removes the Chooser, and returns the choice.
 */

char Chooser::Choose () {
    bs_1->SetValue(false);
    bs_2->SetValue(false);
                                                        bs_3->SetValue(false);
    cancel->SetValue(false);

    PopUp();

    int first = false;
    int second = false;
                                                        int third = false;
    int cancelled = false;
    while (!first && !second && !third && !cancelled) {
                        Event e;
                        Read(e);
                        if (e.target == button_1 || e.target == button_2 ||
                                                            e.target == button_3 || e.target == cancelbutton) {
                          e.target->Handle(e);
                        }
                        bs_1->GetValue(first);
                        bs_2->GetValue(second);
                                                        bs_3->GetValue(third);
                    cancel->GetValue(cancelled);
    }

    Disappear();

    char answer = 'c';
    answer = first ? 'f' : answer;
    answer = second ? 's' : answer;
                                                        answer = third ? 't' : answer;

    return answer;
}

/*
 * Init composes Chooser's view with boxes, glue, and frames.
 */

void Chooser::Init () {
    SetClassName("Chooser");

    HBox* buttons = new HBox;
    buttons->Insert(new HGlue);
    buttons->Insert(button_1);
    buttons->Insert(new HGlue);
    buttons->Insert(button_2);
    buttons->Insert(new HGlue);
                                                        buttons->Insert(button_3);
                                                        buttons->Insert(new HGlue);
    buttons->Insert(cancelbutton);
    buttons->Insert(new HGlue);

    VBox* vbox = new VBox;
    vbox->Align(Center);
    vbox->Insert(new VGlue);
    vbox->Insert(warning);
    vbox->Insert(new VGlue);
    vbox->Insert(message);
    vbox->Insert(new VGlue);
```

```
    vbox->Insert(buttons);
    vbox->Insert(new VGlue);

    Insert(new Frame(vbox, 2));
}

/*
 * Reconfig pads Chooser's shape to make the view look less crowded.
 */

void Chooser::Reconfig () {
    DialogBox::Reconfig();
    Font* font = output->GetFont();
    shape->width += 4 * font->Width("mmmm");
    shape->height += 4 * font->Height();
}
```

drawing.h

```
#ifndef drawing_h
#define drawing_h

#include <InterViews/Std/stdio.h>
#include <InterViews/defs.h>
#include <InterViews/Graphic/classes.h>
#include <InterViews/Std/stream.h>

// Declare imported types.

class BSplineSelection;
class CenterList;
class CommentList;
class DrawingView;
class EdgeList;
class Edge;
class EllipseSelection;
class Graphic;
class GroupList;
class IBrush;
class IBrushList;
class IColor;
class IColorList;
class Idraw;
class IFont;
class IFontList;
class IPattern;
class IPatternList;
class Page;
class PictSelection;
class RectSelection;
class Selection;
class SelectionList;
class State;
class Transformer;
class TextBuffer;
class booleanList;
class Vertex;
class VertexList;
class VertexNode;
class Operator;
class TextSelection;
class World;

// A Drawing contains the user's picture and provides the interface
// through which editing operations modify it.

class Drawing {
public:

    Drawing(double w, double h, double b);
    ~Drawing();

    boolean GetLandscape();
    Page* GetPage();
    void GetPictureTT(Transformer&);
```

```
SelectionList* GetSelectionList();

boolean Writable(const char*);
boolean Exists(const char*);
void ClearPicture();
boolean ReadPicture(const char*, State*);
boolean PrintPicture(const char*, State*);
boolean WritePicture(const char*, State*);
SelectionList* ReadClipboard(State*);
void WriteClipboard();

void GetBox(Coord&, Coord&, Coord&, Coord&);
IBrushList* GetBrush();
CenterList* GetCenter();
GroupList* GetChildren();
IColorList* GetFgColor();
IColorList* GetBgColor();
SelectionList* GetDuplicates();
booleanList* GetFillBg();
IFontList* GetFont();
int GetNumberOfGraphics();
GroupList* GetParent();
IPatternList* GetPattern();
SelectionList* GetPrevs();
SelectionList* GetSelections();

Selection* PickSelectionIntersecting(Coord, Coord);
Selection* PickSelectionShapedBy(Coord, Coord);
SelectionList* PickSelectionsWithin(Coord, Coord, Coord, Coord);

void Clear();
void Extend(Selection*);
void Extend(SelectionList*);
void Grasp(Selection*);
                                        void Grasp(SelectionList*);

void Select(Selection*);
void Select(SelectionList*);
void SelectAll();

void Move(float, float, DrawingView*);
void Scale(float, float);
void Stretch(float, Alignment);
void Rotate(float);
void Align(Alignment, Alignment);
void AlignToGrid();

void SetBrush(IBrush*);
void SetBrush(IBrushList*);
void SetCenter(CenterList*);
void SetFgColor(IColor*);
void SetFgColor(IColorList*);
void SetBgColor(IColor*);
void SetBgColor(IColorList*);
void SetFillBg(boolean);
void SetFillBg(booleanList*);
void SetFont(IFont*);
void SetFont(IFontList*);
void SetPattern(IPattern*);
void SetPattern(IPatternList*);

void Append();
void Group(GroupList*);
void InsertAfterPrev(SelectionList*);
void Prepend();
void Remove();
void Replace(Selection*, Selection*);
void Sort();
void Ungroup(GroupList*);


// DFD specific functions

Vertex* SetEndptsInOperator(Coord&, Coord&, Coord&, Coord&);
void OperatorAppend(EllipseSelection*,int = 0);
void OperatorAppend(RectSelection*,int = 0);
void DataFlowSplineAppend(BSplineSelection*, Vertex*,Vertex*);
```

```
  void  AddTextToSelectionList();
  void  AddSelfLoopsToSelectionList();
  void  ReplaceAssociatedObjects(Vertex*, DrawingView*);

  void  TriggerAppend (TextSelection*, EllipseSelection*);
  void  WritePSDLSpec(FILE*);
  void WritePSDLImpl(FILE*);
  void WritePSDLGraph(FILE*);
  void  WriteDFDFiles(char*, const char*);
  void  WritePSDLForAllOperators(char*, const char*);


// Eagle additions for .graph file format change
  void TagOperator(Vertex*,char*);
  void GenerateAda(FILE*,Vertex*);
  void CommentAppend(TextSelection*);
  void SetDrawingView(DrawingView*);
  void SetState(State *s);
  void SetIdraw(Idraw *i) {idraw = i;}
  EdgeList *GetEdgeList() {return el;}
// end changes


  void ReadDFDFiles(char*, const char*);
  void AddAllAssociatedItems(SelectionList*);

protected:

  int NumberOfGraphics(PictSelection*);

// DFD specific functions
  void CheckForComposites(char*, const char*);

  void AddInputSelection(Selection*);
  IColor *ReadFgColor(FILE*);
  IColor *ReadBgColor(FILE*);
  void WriteFgColor(FILE*,IColor*);
  void WriteBgColor(FILE*,IColor*);
  IFont *ReadFont(FILE*);
  void WriteFont(FILE*, IFont*);
  void ReadPoint(FILE*,int,Coord&,Coord&);
  void WritePoint(FILE*,Transformer*,Coord,Coord);
  void WriteTS(TextSelection*, FILE*);
  TextSelection *ReadTS(ClassId, FILE*);
  void WriteDFDInfo(FILE*);
  void ReadGraphEdge(FILE*);
  void ReadGraphVertex(FILE*,int);
  void ReadDFDInfo(FILE*);

  void WriteEdges(FILE*);
  void WriteVertices(FILE*);
  void WritePSDLConstraints(FILE*);

  void WriteGraphEdge(FILE*,Edge*);
  void WriteGraphVertex(FILE*,Vertex*);

  Vertex* EndptsInOperator(Coord&, Coord&);
  void FindOperatorIntersection(Coord&, Coord&, Coord, Coord);
  void FindTerminatorIntersection(Coord&, Coord&, Coord, Coord);
  void ReplaceInputDFSplines(Vertex*, DrawingView*);
  void ReplaceOutputDFSplines(Vertex*, DrawingView*);
  void ReplaceLabel(Selection*, TextSelection*);
  void ReplaceLatency(Selection*, TextSelection*);

  char* clipfilename;                    // filename under which to store clippings
  Page* page;                                 // draws picture
  PictSelection* picture;     // stores picture
  CommentList *cl;                 // list of comments
  SelectionList* sl;                     // lists picked Selections
  VertexList* ol;             // list of operators drawn
  EdgeList *el;

// eagle changes for Shing editor
  State *state;
  Idraw *idraw;
  DrawingView *drawingview;
```

```
};

inline void Drawing::SetState(State *s) {
  state = s;
}

inline void Drawing::SetDrawingView(DrawingView *dv) {
  drawingview = dv;
}

#endif
```

drawing.c

```
#include "commentlist.h"
#include "dfd_defs.h"
#include "dfdclasses.h"
#include "drawingview.h"
#include "drawing.h"
#include "idraw.h"
#include "ipaint.h"
#include "istring.h"
#include "listboolean.h"
#include "listcenter.h"
#include "listchange.h"
#include "listgroup.h"
#include "listibrush.h"
#include "listicolor.h"
#include "listifont.h"
#include "listipattern.h"
#include "listselectn.h"
#include "mapipaint.h"
#include "page.h"
#include "psdlcomp.h"
#include "psdllists.h"
#include "slellipses.h"
#include "slpict.h"
#include "slpolygons.h"
#include "slsplines.h"
#include "sltext.h"
#include "state.h"
#include <InterViews/Graphic/polygons.h>
#include <InterViews/defs.h>
#include <InterViews/regexp.h>
#include <InterViews/shape.h>
#include <InterViews/perspective.h>
#include <InterViews/textbuffer.h>
#include <InterViews/transformer.h>
#include <InterViews/Std/os/fs.h>
#include <InterViews/Std/stdio.h>
#define KERNEL
#include <stdlib.h>
#include <sys/file.h>                          /* define constants for access call */
#define _POSIX_SOURCE
#include <math.h>
/* extern "C"
{
 extern char *getenv( char*);
} */

// Drawing creates the page, selection list, and clipboard filename.

Drawing::Drawing (double w, double h, double b) {
  const char* home = (home = getenv("HOME")) ? home : ".";
  const char* name = ".clipboard";
  clipfilename = new char[strlen(home) + 1 + strlen(name) + 1];
  strcpy(clipfilename, home);
  strcat(clipfilename, "/");
  strcat(clipfilename, name);
  page = new Page(w, h, b);
  picture = page->GetPicture();
  sl = new SelectionList;
  cl = new CommentList;
  ol = new VertexList;
  el = new EdgeList;
}
```

```
// ~Drawing frees storage allocated for the clipboard filename, page,
// and selection list.

Drawing::~Drawing ()
 {
  delete clipfilename;
  delete page;
  delete sl;
  delete cl;
  delete ol;
  delete el;
 }

// Define access functions to return attributes.

boolean Drawing::GetLandscape ()
 {
  Transformer* t = page->GetTransformer();
  return t ? t->Rotated90() : false;
 }

Page* Drawing::GetPage ()
{
  return page;
 }

void Drawing::GetPictureTT (Transformer& t)
 {
  picture->TotalTransformation(t);
 }

SelectionList* Drawing::GetSelectionList ()
 {
  return sl;
 }

// Writable returns true only if the given drawing is writable.

boolean Drawing::Writable (const char* path)
{
 return (access(path, W_OK) >= 0);
 }

// Exists returns true only if the drawing already exists.

boolean Drawing::Exists (const char* path)
{
  return (access(path, F_OK) >= 0);
}

// ClearPicture deletes the old picture and creates a new empty
// picture.
// Modified 10/20/94 by C.S.Eagle

void Drawing::ClearPicture ()
{
  sl->DeleteAll();
  page->SetPicture(nil);
  picture = page->GetPicture();

 ol->DeleteAll();
 el->DeleteAll();
}

// ReadPicture reads a new picture and replaces the old picture with
// the new picture if the read succeeds.

boolean Drawing::ReadPicture (const char* path, State* state)
{
 boolean successful = false;
 if (path != nil)
  {
   FILE* stream = fopen(path, "r");
   if (stream != nil)
    {
                PictSelection* newpic = new PictSelection(stream, state);
```

```
        fclose(stream);
        if (newpic->Valid())
          {
                        sl->DeleteAll();
                        page->SetPicture(newpic);
                        picture = page->GetPicture();
                        successful = true;
                        }
        else
          {
                        delete newpic;
                        fprintf(stderr, "Drawing: input error in reading %s\n", path);
                        }
                      }
    }
    return successful;
}

// PrintPicture prints the current picture by writing it through a
// pipe to a print command.

boolean Drawing::PrintPicture (const char* cmd, State* state)
 {
   boolean successful = false;
   if (cmd != nil)
     {
                FILE* stream = popen(cmd, "w");
                if (stream != nil)
       {
                  successful = picture->WritePicture(stream, state, true);
                  pclose(stream);
                 }
     }
   return successful;
 }

// WritePicture writes the current picture to a file.

boolean Drawing::WritePicture (const char* path, State* state)
  {
   boolean successful = false;
   if (path != nil)
     {
                FILE* stream = fopen(path, "w");
                if (stream != nil)
       {
                  successful = picture->WritePicture(stream, state, true);
                  fclose(stream);
                 }
     }
   return successful;
  }

// ReadClipboard returns copies of the Selections within the clipboard
// file in a newly allocated list.

SelectionList* Drawing::ReadClipboard (State* state)
 {
   SelectionList* sl = new SelectionList;
   FILE* stream = fopen(clipfilename, "r");
   if (stream != nil)
     {
                PictSelection* newpic = new PictSelection(stream, state);
                fclose(stream);
                if (newpic->Valid())
       {
                  newpic->Propagate();
                  for (newpic->First(); !newpic->AtEnd(); newpic->RemoveCur())
         {
                                                    Selection* child = (Selection*) newpic->GetCurrent();
                                                    sl->Append(new SelectionNode(child));
           }
                 }
                 delete newpic;
     }
   else
```

```
    {
                fprintf(stderr, "Drawing: can't open %s\n", clipfilename);
    }
  return sl;
 }

// WriteClipboard writes the picked Selections to the clipboard file,
// overwriting its previous contents.

void Drawing::WriteClipboard ()
 {
  FILE* stream = fopen(clipfilename, "w");
  if (stream != nil)
    {
                PictSelection* newpic = new PictSelection;
                for (sl->First(); !sl->AtEnd(); sl->Next())
      {
                  Graphic* copy = sl->GetCur()->GetSelection()->Copy();
                  newpic->Append(copy);
                 }
                newpic->WritePicture(stream, nil, false);
                fclose(stream);
                delete newpic;
    }
  else
    {
                fprintf(stderr, "Drawing: can't open %s\n", clipfilename);
    }
 }

// GetBox gets the smallest box bounding all the Selections.

void Drawing::GetBox (Coord& l, Coord& b, Coord& r, Coord& t)
 {
  BoxObj btotal;
  BoxObj bselection;

  if (sl->Size() >= 1)
    {
                sl->First()->GetSelection()->GetBox(btotal);
                for (sl->Next(); !sl->AtEnd(); sl->Next())
      {
                  sl->GetCur()->GetSelection()->GetBox(bselection);
                  btotal = btotal + bselection;
                 }
                l = btotal.left;
                b = btotal.bottom;
                r = btotal.right;
                t = btotal.top;
    }
 }

// GetBrush returns the Selections' brush attributes in a newly
// allocated list.

IBrushList* Drawing::GetBrush ()
 {
  IBrushList* brushlist = new IBrushList;
  for (sl->First(); !sl->AtEnd(); sl->Next())
    {
                IBrush* brush = (IBrush*) sl->GetCur()->GetSelection()->GetBrush();
                brushlist->Append(new IBrushNode(brush));
    }
  return brushlist;
 }

// GetCenter returns the Selections' centers in a newly allocated
// list. It converts the centers from window coordinates to picture
// coordinates because only these coordinates will remain constant.

CenterList* Drawing::GetCenter ()
 {
  CenterList* centerlist = new CenterList;
  Transformer t;
  picture->TotalTransformation(t);
  for (sl->First(); !sl->AtEnd(); sl->Next())
```

```
    {
                float wincx, wincy, cx, cy;

                sl->GetCur()->GetSelection()->GetCenter(wincx, wincy);
                t.InvTransform(wincx, wincy, cx, cy);
                centerlist->Append(new CenterNode(cx, cy));
    }
  return centerlist;
  }

// GetChildren returns the Selections and their children, if any, in a
// newly allocated list.

GroupList* Drawing::GetChildren ()
 {
  GroupList* grouplist = new GroupList;
  for (sl->First(); !sl->AtEnd(); sl->Next())
    {
                PictSelection* parent = (PictSelection*) sl->GetCur()->GetSelection();
                boolean haschildren = parent->HasChildren();
                SelectionList* children = new SelectionList;
                if (haschildren)
      {
                  for (parent->First(); !parent->AtEnd(); parent->Next())
        {
                                                        Selection* child = parent->GetCurrent();
                                                        children->Append(new SelectionNode(child));
          }
        }
                grouplist->Append(new GroupNode(parent, haschildren, children));
                delete children;
    }
  return grouplist;
  }

// GetFgColor returns the Selections' FgColor attributes in a newly
// allocated list.

IColorList* Drawing::GetFgColor ()
 {
  IColorList* fgcolorlist = new IColorList;
  for (sl->First(); !sl->AtEnd(); sl->Next())
    {
                IColor* fgcolor = (IColor*) sl->GetCur()->GetSelection()->GetFgColor();
                fgcolorlist->Append(new IColorNode(fgcolor));
    }
  return fgcolorlist;
  }

// GetBgColor returns the Selections' BgColor attributes in a newly
// allocated list.

IColorList* Drawing::GetBgColor ()
 {
  IColorList* bgcolorlist = new IColorList;
  for (sl->First(); !sl->AtEnd(); sl->Next())
    {
                IColor* bgcolor = (IColor*) sl->GetCur()->GetSelection()->GetBgColor();
                bgcolorlist->Append(new IColorNode(bgcolor));
    }
  return bgcolorlist;
}

// GetDuplicates duplicates the Selections, offsets them by one grid
// spacing, and returns them in a newly allocated list.

SelectionList* Drawing::GetDuplicates ()
 {
  int offset = round(page->GetGridSpacing() * points);
  SelectionList* duplicates = new SelectionList;
  for (sl->First(); !sl->AtEnd(); sl->Next())
    {
                Selection* dup = (Selection*) sl->GetCur()->GetSelection()->Copy();
                dup->Translate(offset, offset);
                duplicates->Append(new SelectionNode(dup));
    }
```

```
  return duplicates;
 }

// GetFillBg returns the Selections' fillbg attributes in a newly
// allocated list.

booleanList* Drawing::GetFillBg ()
 {
  booleanList* fillbglist = new booleanList;
  for (sl->First(); !sl->AtEnd(); sl->Next())
   {
                boolean fillbg = sl->GetCur()->GetSelection()->BgFilled();
                fillbglist->Append(new booleanNode(fillbg));
   }
  return fillbglist;
 }

// GetFont returns the Selections' Font attributes in a newly
// allocated list.

IFontList* Drawing::GetFont ()
 {
  IFontList* fontlist = new IFontList;
  for (sl->First(); !sl->AtEnd(); sl->Next())
   {
                IFont* font = (IFont*) sl->GetCur()->GetSelection()->GetFont();
                fontlist->Append(new IFontNode(font));
   }
  return fontlist;
  }

// GetNumberOfGraphics returns the number of graphics in the
// Selections.

int Drawing::GetNumberOfGraphics ()
  {
   int num = 0;
   for (sl->First(); !sl->AtEnd(); sl->Next())
    {
                Selection* s = sl->GetCur()->GetSelection();
                if (s->HasChildren())
                  num += NumberOfGraphics((PictSelection*) s);
     else
                  ++num;
    }
   return num;
  }

// GetParent returns the Selections and their new parent in a newly
// allocated list if there are enough Selections to form a Group.

GroupList* Drawing::GetParent ()
 {
  GroupList* grouplist = new GroupList;
  if (sl->Size() >= 2)
   {
                PictSelection* parent = new PictSelection;
                boolean haschildren = true;
                grouplist->Append(new GroupNode(parent, haschildren, sl));
   }
  return grouplist;
 }

// GetPattern returns the Selections' Pattern attributes in a newly
// allocated list.

IPatternList* Drawing::GetPattern ()
 {
  IPatternList* patternlist = new IPatternList;
  for (sl->First(); !sl->AtEnd(); sl->Next())
   {
                IPattern* pattern =
                  (IPattern*) sl->GetCur()->GetSelection()->GetPattern();
                patternlist->Append(new IPatternNode(pattern));
   }
  return patternlist;
```

```
}

// GetPrevs returns the Selections' predecessors within the picture in
// a newly allocated list.

SelectionList* Drawing::GetPrevs ()
 {
  SelectionList* prevlist = new SelectionList;
  for (sl->First(); !sl->AtEnd(); sl->Next())
   {
                picture->SetCurrent(sl->GetCur()->GetSelection());
                Selection* prev = picture->Prev();
                prevlist->Append(new SelectionNode(prev));
   }
  return prevlist;
  }

// GetSelections returns the Selections in a newly allocated list.

SelectionList* Drawing::GetSelections ()
 {
  SelectionList* newsl = new SelectionList;
  for (sl->First(); !sl->AtEnd(); sl->Next())
   {
                Selection* s = sl->GetCur()->GetSelection();
                newsl->Append(new SelectionNode(s));
   }
  return newsl;
  }

// PickSelectionIntersecting returns the last Selection intersecting a
// box around the given point.

Selection* Drawing::PickSelectionIntersecting (Coord x, Coord y)
 {
  const int SLOP = 2;
  BoxObj pickpoint(x - SLOP, y - SLOP, x + SLOP, y + SLOP);
  return picture->LastSelectionIntersecting(pickpoint);
  }

// PickSelectionShapedBy returns the last Selection shaped by a point
// close to the given point.

Selection* Drawing::PickSelectionShapedBy (Coord x, Coord y)
 {
  const float SLOP = 6.;
  for (picture->Last(); !picture->AtEnd(); picture->Prev())
   {
                Selection* pick = picture->GetCurrent();
                if (pick->ShapedBy(x, y, SLOP))
                  return pick;
   }
  return nil;
 }

// PickSelectionsWithin returns all the Selections within the given
// box.

SelectionList* Drawing::PickSelectionsWithin (Coord l, Coord b, Coord r, Coord t)
 {
  Selection** picks = nil;
  int numpicks = picture->SelectionsWithin(BoxObj(l, b, r, t), picks);
  SelectionList* picklist = new SelectionList;
  for (int i = 0; i < numpicks; i++)
                if (!picklist->Find(picks[i]))
                   picklist->Append(new SelectionNode(picks[i]));
  delete picks;
  return picklist;
  }

// Clear empties the SelectionList.

void Drawing::Clear ()
 {
  sl->DeleteAll();
  }
```

```
// Extend extends the SelectionList to include the picked Selection
// unless it's already there, in which case it removes the Selection.

void Drawing::Extend (Selection* pick)
 {
  if (!sl->Find(pick))
     sl->Append(new SelectionNode(pick));
  else
                sl->DeleteCur();
 }

// Extend extends the SelectionList to include the picked Selections
// unless they're already there, in which case it removes them.

void Drawing::Extend (SelectionList* picklist)
 {
  for (picklist->First(); !picklist->AtEnd(); picklist->Next())
    {
                Selection* pick = picklist->GetCur()->GetSelection();
                Extend(pick);
    }
 }

// Grasp selects the picked Selection only if the SelectionList does
// not already include it.

void Drawing::Grasp (Selection* pick)
 {
  if (!sl->Find(pick))
                Select(pick);
 }

// Grasp selects the list of picked  Selections only if the SelectionList
// does not already include any of the selections.

void Drawing::Grasp (SelectionList* picklist)
 {
  boolean found = false;
  for (picklist->First(); !picklist->AtEnd(); picklist->Next())
    {
                if (sl->Find(picklist->GetCur()->GetSelection()))
      {
                   found = true;
                   exit(0);
      }
    }
  if (!found)
     Select(picklist);
 }

// Select selects the picked Selection.

void Drawing::Select (Selection* pick)
 {
  sl->DeleteAll();
  sl->Append(new SelectionNode(pick));
 }

// Select selects the picked Selections.

void Drawing::Select (SelectionList* picklist)
 {
  sl->DeleteAll();
  for (picklist->First(); !picklist->AtEnd(); picklist->Next())
    {
                Selection* pick = picklist->GetCur()->GetSelection();
                sl->Append(new SelectionNode(pick));
    }
  }

// SelectAll selects all of the Selections in the picture.

void Drawing::SelectAll ()
 {
  sl->DeleteAll();
```

```
  for (picture->First(); !picture->AtEnd(); picture->Next())
   {
                Selection* pick = picture->GetCurrent();
                sl->Append(new SelectionNode(pick));
   }
  }

// Move translates the Selections.

void Drawing::Move (float xdisp, float ydisp, DrawingView* dv)
 {
  // add labels, mets, and selfloops and their labels to the list of selections
  // to be moved if the operator they are attached to is to be moved.
  // These objects can be translated along with the operator - data flows
  // cannot be translated, they must be modified

  AddTextToSelectionList();
  AddSelfLoopsToSelectionList();
  ClassId cid;
  int sl_size = sl->Size();
  for (int index = 0; index < sl_size; ++index)
   {
   Selection* s = sl->Index(index)->GetSelection();
   cid = s->GetClassId();
   if (cid == OPERATOR || cid == TERMINATOR || cid == COMMENT ||
                                                    cid == LABEL_OP || cid == MET_OP ||
                cid == LABEL_SL || cid == SELFLOOP || (sl_size == 1 &&
                                                    (cid == LABEL_DF || cid == LAT_DF)))
    {
                s->Translate(xdisp, ydisp);
                if (cid == OPERATOR || cid == TERMINATOR)
     {
                  SelectionList* temp = GetSelections();
                  ReplaceAssociatedObjects((Vertex*) s->GetOwner(), dv);
                  Select(temp);
     }
    }
   }

  // only redraw the screen once to make the transition look smooth

  dv->Draw();
 }

// Scale scales the Selections about their centers.

void Drawing::Scale (float xscale, float yscale)
 {
  for (sl->First(); !sl->AtEnd(); sl->Next())
   {
   Selection* s = sl->GetCur()->GetSelection();
   float cx, cy;
   s->GetCenter(cx, cy);
   s->Scale(xscale, yscale, cx, cy);
   }
 }

// Stretch stretches the Selections while keeping the given side
// fixed.

void Drawing::Stretch (float stretch, Alignment side)
 {
  for (sl->First(); !sl->AtEnd(); sl->Next())
   {
   Selection* s = sl->GetCur()->GetSelection();
   float l, b, r, t;
   s->GetBounds(l, b, r, t);
   switch (side)
    {
    case Left:
     s->Scale(stretch, 1, r, t);
                 break;
                case Bottom:
                 s->Scale(1, stretch, r, t);
                 break;
                case Right:
```

```
                    s->Scale(stretch, 1, l, b);
                    break;
                  case Top:
                    s->Scale(1, stretch, l, b);
                    break;
      default:
                    fprintf(stderr, "inappropriate enum passed to Drawing::Stretch\n");
                    break;
                  }
  }
}

// Rotate rotates the Selections about their centers.

void Drawing::Rotate (float angle)
 {
  for (sl->First(); !sl->AtEnd(); sl->Next())
   {
    Selection* s = sl->GetCur()->GetSelection();
    float cx, cy;
    s->GetCenter(cx, cy);
    s->Rotate(angle, cx, cy);
   }
}

// Align either aligns up all of the Selections or abuts all of them
// side to side, depending on whether the moving Selection's side or
// center aligns with the fixed Selection's same side or center.

void Drawing::Align (Alignment falign, Alignment malign)
 {
  if (falign == malign)
   {
                Selection* stays = sl->First()->GetSelection();
                for (sl->Next(); !sl->AtEnd(); sl->Next())
     {
                  Selection* moves = sl->GetCur()->GetSelection();
                  stays->Align(falign, moves, malign);
                 }
   }
  else
   {
                Selection* stays = sl->First()->GetSelection();
                for (sl->Next(); !sl->AtEnd(); sl->Next())
     {
                  Selection* moves = sl->GetCur()->GetSelection();
                  stays->Align(falign, moves, malign);
                  stays = moves;
     }
   }
 }

// AlignToGrid aligns the Selections' lower left corners to the
// nearest grid point.

void Drawing::AlignToGrid ()
 {
  boolean gravity = page->GetGridGravity();
  page->SetGridGravity(true);
  Transformer t;
  picture->TotalTransformation(t);

  for (sl->First(); !sl->AtEnd(); sl->Next())
   {
                Selection* s = sl->GetCur()->GetSelection();
                float l, b, dummy;
                s->GetBounds(l, b, dummy, dummy);
                Coord nl = round(l);
                Coord nb = round(b);
                page->Constrain(nl, nb);
                float x0, y0, x1, y1;
                t.InvTransform(l, b, x0, y0);
                t.InvTransform(float(nl), float(nb), x1, y1);
                s->Translate(x1 - x0, y1 - y0);
   }
```

```
    page->SetGridGravity(gravity);
  }

// SetBrush sets the Selections' brush attributes with the given brush
// attribute.

void Drawing::SetBrush (IBrush* brush)
  {
    for (sl->First(); !sl->AtEnd(); sl->Next())
      {
                    sl->GetCur()->GetSelection()->SetBrush(brush);
      }
  }

// SetBrush sets each Selection's brush attribute with the
// corresponding brush attribute in the provided list.

void Drawing::SetBrush (IBrushList* brushlist)
  {
    for (sl->First(), brushlist->First(); !sl->AtEnd() && !brushlist->AtEnd();
                    sl->Next(), brushlist->Next())
      {
                    IBrush* brush = brushlist->GetCur()->GetBrush();
                    sl->GetCur()->GetSelection()->SetBrush(brush);
      }
  }

// SetCenter centers each of the Selections over the corresponding
// position in the provided list.  It expects the passed postions to
// in picture coordinates, not window coordinates.

void Drawing::SetCenter (CenterList* centerlist)
  {
    Transformer t;
    picture->TotalTransformation(t);
    for (sl->First(), centerlist->First();
                    !sl->AtEnd() && !centerlist->AtEnd();
                    sl->Next(), centerlist->Next())
      {
                    float winoldcx, winoldcy, oldcx, oldcy;
                    float newcx = centerlist->GetCur()->GetCx();
                    float newcy = centerlist->GetCur()->GetCy();
                    Selection* s = sl->GetCur()->GetSelection();

                    s->GetCenter(winoldcx, winoldcy);
                    t.InvTransform(winoldcx, winoldcy, oldcx, oldcy);
                    s->Translate(newcx - oldcx, newcy - oldcy);
      }
  }

// SetFgColor sets the Selections' foreground color attributes with
// the given color attribute.

void Drawing::SetFgColor (IColor* fgcolor)
  {
    for (sl->First(); !sl->AtEnd(); sl->Next())
      {
                    Selection* s = sl->GetCur()->GetSelection();
                    IColor* bgcolor = (IColor*) s->GetBgColor();
                    s->SetColors(fgcolor, bgcolor);
      }
  }

// SetFgColor sets the Selections' foreground color attributes with
// the corresponding color attributes in the provided list.

void Drawing::SetFgColor (IColorList* fgcolorlist)
  {
    for (sl->First(), fgcolorlist->First();
                    !sl->AtEnd() && !fgcolorlist->AtEnd();
                    sl->Next(), fgcolorlist->Next())
      {
                    Selection* s = sl->GetCur()->GetSelection();
                    IColor* fgcolor = fgcolorlist->GetCur()->GetColor();
                    IColor* bgcolor = (IColor*) s->GetBgColor();
                    s->SetColors(fgcolor, bgcolor);
```

```
    }
  }

// SetBgColor sets the Selections' background color attributes with
// the given color attribute.

void Drawing::SetBgColor (IColor* bgcolor)
 {
  for (sl->First(); !sl->AtEnd(); sl->Next())
    {
                Selection* s = sl->GetCur()->GetSelection();
                IColor* fgcolor = (IColor*) s->GetFgColor();
                s->SetColors(fgcolor, bgcolor);
    }
  }

// SetBgColor sets the Selections' background color attributes with
// the corresponding color attributes in the provided list.

void Drawing::SetBgColor (IColorList* bgcolorlist)
 {
  for (sl->First(), bgcolorlist->First();
                !sl->AtEnd() && !bgcolorlist->AtEnd();
                sl->Next(), bgcolorlist->Next())
    {
                Selection* s = sl->GetCur()->GetSelection();
                IColor* fgcolor = (IColor*) s->GetFgColor();
                IColor* bgcolor = bgcolorlist->GetCur()->GetColor();
                s->SetColors(fgcolor, bgcolor);
    }
  }

// SetFillBg sets the Selections' fillbg attributes with the given
// fillbg attribute.

void Drawing::SetFillBg (boolean fillbg)
 {
  for (sl->First(); !sl->AtEnd(); sl->Next())
                sl->GetCur()->GetSelection()->FillBg(fillbg);
  }

// SetFillBg sets each Selection's fillbg attribute with the
// corresponding fillbg attribute in the provided list.

void Drawing::SetFillBg (booleanList* fillbglist)
 {
  for (sl->First(), fillbglist->First();
                !sl->AtEnd() && !fillbglist->AtEnd();
                sl->Next(), fillbglist->Next())
    {
                boolean fillbg = fillbglist->GetCur()->GetBoolean();
                sl->GetCur()->GetSelection()->FillBg(fillbg);
    }
  }

// SetFont sets the Selections' font attributes with the given font
// attribute.

void Drawing::SetFont (IFont* font)
 {
  for (sl->First(); !sl->AtEnd(); sl->Next())
    {
                Selection* s = sl->GetCur()->GetSelection();
                s->SetFont(font);
    }
  }

// SetFont sets each Selection's font attribute with the corresponding
// font attribute in the provided list.

void Drawing::SetFont (IFontList* fontlist)
 {
  for (sl->First(), fontlist->First(); !sl->AtEnd() && !fontlist->AtEnd();
                sl->Next(), fontlist->Next())
    {
                IFont* font = fontlist->GetCur()->GetFont();
```

```
                Selection* s = sl->GetCur()->GetSelection();
                s->SetFont(font);
    }
  }

// SetPattern sets the Selections' pattern attributes with the given
// pattern attribute.

void Drawing::SetPattern (IPattern* pattern)
  {
   for (sl->First(); !sl->AtEnd(); sl->Next())
     {
                sl->GetCur()->GetSelection()->SetPattern(pattern);
     }
  }

// SetPattern sets each Selection's pattern attribute with the
// corresponding pattern attribute in the provided list.

void Drawing::SetPattern (IPatternList* patternlist)
  {
   for (sl->First(), patternlist->First();
                !sl->AtEnd() && !patternlist->AtEnd();
                sl->Next(), patternlist->Next())
     {
                IPattern* pattern = patternlist->GetCur()->GetPattern();
                sl->GetCur()->GetSelection()->SetPattern(pattern);
     }
  }

// Append appends the Selections to the picture.

void Drawing::Append ()
  {
   for (sl->First(); !sl->AtEnd(); sl->Next())
     {
                Selection* s = sl->GetCur()->GetSelection();
                picture->Append(s);
     }
  }

// Group groups each parent's children, if any, under their parent and
// returns the resulting Selections in the SelectionList.

void Drawing::Group (GroupList* grouplist)
  {
   if (grouplist->Size() >= 1)
     {
                sl->DeleteAll();
                for (grouplist->First(); !grouplist->AtEnd(); grouplist->Next())
       {
                  GroupNode* gn = grouplist->GetCur();
                  PictSelection* parent = gn->GetParent();
                  boolean haschildren = gn->GetHasChildren();
                  SelectionList* children = gn->GetChildren();
                  SelectionList* childrengs = gn->GetChildrenGS();
                  if (haschildren)
         {
                                                        for (children->First(), childrengs->First();
                                                           !children->AtEnd() && !childrengs->AtEnd();
                                                           children->Next(), childrengs->Next())
                                                        {
                                                         Graphic* child = children->GetCur()->GetSelection();
                                                         Graphic* childgs = childrengs->GetCur()->GetSelection();
                                                         *child = *childgs;
                                                         picture->SetCurrent(child);
                                                         picture->Remove(child);
                                                         parent->Append(child);
                                                        }
                                                       picture->InsertBeforeCur(parent);
                  }
                  sl->Append(new SelectionNode(parent));
                }
     Sort();
     }
  }
```

```
// InsertAfterPrev inserts each Selection after its corresponding
// predecessor in the provided list.

void Drawing::InsertAfterPrev (SelectionList* prevlist)
 {
  for (sl->First(), prevlist->First(); !sl->AtEnd() && !prevlist->AtEnd();
                    sl->Next(), prevlist->Next())
   {
                    Selection* prev = prevlist->GetCur()->GetSelection();
                    picture->SetCurrent(prev);
                    Selection* s = sl->GetCur()->GetSelection();
                    picture->InsertAfterCur(s);
   }
 }

// Prepend prepends the Selections to the picture.

void Drawing::Prepend () {
  for (sl->Last(); !sl->AtEnd(); sl->Prev()) {
                    Selection* s = sl->GetCur()->GetSelection();
                    picture->Prepend(s);
  }
}

// Remove removes the Selections from the picture.

void Drawing::Remove () {
  SelectionList* sl_2 = GetSelections();
  Clear();
  AddAllAssociatedItems(sl_2);
  for (sl_2->First(); !sl_2->AtEnd(); sl_2->Next())
    picture->Remove(sl_2->GetCur()->GetSelection());
  for (sl_2->First(); !sl_2->AtEnd(); sl_2->Next()) {
                     Selection* s = sl_2->GetCur()->GetSelection();
                     ClassId cid = s ->GetClassId();
    if (s->IsEdgeComponent())
      el->Remove(s);
    else if (s->IsVertexComponent())
      ol->Remove(s);
    else if (s->GetClassId() == COMMENT)
      cl->Remove((TextSelection *)s);
  }
  sl_2->DeleteAll();
}

void Drawing::AddAllAssociatedItems(SelectionList *s) {
  TextSelection *ts;
  EdgeList *edl;
  BSplineSelection *bs;
  for (s->First(); !s->AtEnd(); s->Next()) {
    Selection *sel = s->GetCur()->GetSelection();
    if (sel->GetClassId() == OPERATOR) {
      Operator *op = (Operator *) sel->GetOwner();
      ts = op->GetLabel();
      if (ts && !s->Find(ts))
        s->Append(new SelectionNode(ts));
      ts = op->GetMET();
      if (ts && !s->Find(ts))
        s->Append(new SelectionNode(ts));
      ts = op->GetTag();
      if (ts && !s->Find(ts))
        s->Append(new SelectionNode(ts));
      edl = op->GetInputs();
      for (edl->First(); !edl->AtEnd(); edl->Next()) {
        bs = edl->GetCur()->GetEdge()->GetBSpline();
        if (bs && !s->Find(bs))
          s->Append(new SelectionNode(bs));
                    }
      edl = op->GetOutputs();
      for (edl->First(); !edl->AtEnd(); edl->Next()) {
        bs = edl->GetCur()->GetEdge()->GetBSpline();
        if (bs && !s->Find(bs))
          s->Append(new SelectionNode(bs));
                    }
    }
```

```
    else if ((sel->GetClassId() == DATAFLOW_SPLINE) ||
          (sel->GetClassId() == SELFLOOP)) {
      Edge *e = (Edge *) sel->GetOwner();
      ts = e->GetLabel();
      if (ts && !s->Find(ts))
        s->Append(new SelectionNode(ts));
      ts = e->GetLatency();
      if (ts && !s->Find(ts))
        s->Append(new SelectionNode(ts));
    }
  }
}

// Replace replaces a Selection in the picture with a Selection not in it.

void Drawing::Replace (Selection* replacee, Selection* replacer) {
  if (replacee->IsEdgeComponent())
    el->Replace(replacee,replacer);
  else if (replacee->IsVertexComponent())
    ol->Replace(replacee,replacer);
  picture->SetCurrent(replacee);
  picture->Remove(replacee);
  picture->InsertBeforeCur(replacer);
}

// Sort sorts the Selections so they occur in the same order as they
// do in the picture.

void Drawing::Sort () {
  if (sl->Size() >= 2)
                for (picture->First(); !picture->AtEnd(); picture->Next()) {
                  Selection* g = picture->GetCurrent();
                  if (sl->Find(g)) {
                                                        SelectionNode* s = sl->GetCur();
                                                        sl->RemoveCur();
                                                        sl->Append(s);

                  }
                }
}

// Ungroup replaces all Selections which contain children with their
// children and returns the resulting Selections in the SelectionList.

void Drawing::Ungroup (GroupList* grouplist)
 {
  if (grouplist->Size() >= 1)
    {
                sl->DeleteAll();
                for (grouplist->First(); !grouplist->AtEnd(); grouplist->Next())
      {
                  GroupNode* gn = grouplist->GetCur();
                  PictSelection* parent = gn->GetParent();
                  boolean haschildren = gn->GetHasChildren();
                  SelectionList* children = gn->GetChildren();
                  if (haschildren)
        {
                                                        parent->Propagate();
                                                        picture->SetCurrent(parent);
                                                        for (children->First(); !children->AtEnd(); children->Next())

          {
                                                          Selection* child = children->GetCur()->GetSelection();
                                                          parent->Remove(child);
                                                          picture->InsertBeforeCur(child);
                                                          sl->Append(new SelectionNode(child));
                                                         }
                                                        picture->Remove(parent);

                  }
        else
                                                        sl->Append(new SelectionNode(parent));

                  }
                Sort();
    }
  }

// NumberOfGraphics returns the number of graphics in the picture,
// calling itself recursively to count the number of graphics in
```

```
// subpictures.

int Drawing::NumberOfGraphics (PictSelection* picture)
 {
  int num = 0;
  for (picture->First(); !picture->AtEnd(); picture->Next())
   {
    Selection* s = picture->GetCurrent();
                if (s->HasChildren())
                  num += NumberOfGraphics((PictSelection*) s);
    else
                  ++num;
   }
  return num;
 }

// DFD specific functions

// return ellipse that contains the given points

Vertex* Drawing::EndptsInOperator(Coord& x, Coord& y) {
  Vertex *op = ol->FindOp(x,y);
  if (op) {
                float fx, fy;
                op->GetShape()->GetCenter(fx, fy);
                x = (Coord) fx;
                y = (Coord) fy;
                return op;
  }
  else
    return 0;
}

// Find point where data flow intersects with operator in order to
// draw data flow's endpoints at intersection point with operator
// Changed 2/6/91
// By Patrick D. Barnes
// Description: Fixed bug where new endpoint calculation is hosed when
//  picture is zoomed, resized, or scrolled. Previous code did not
//  adjust OperatorRadius for Zoom.


void Drawing::FindOperatorIntersection(Coord& x, Coord& y,
                  Coord x2, Coord y2) {
  float a00, a01, a10, scale, a20, a21;
  int radius;
  Transformer t;
  GetPictureTT(t);
  t.GetEntries(a00, a01, a10, scale, a20, a21);
  radius = (int) ((float) OperatorRadius * scale);
  if (x != x2 && y != y2)
   {
                double temp1, temp2, h, a, den;
                temp1 = (double) (x2 - x);
                temp2 = (double) (y2 - y);
                den = sqrt((temp1 * temp1) + (temp2 * temp2));
                a = (double) (radius) * (temp1 / den);
                h = (temp2 / temp1) * a;
    x += (Coord) a;
                y += (Coord) h;
   }
  else
   {
                if (x == x2)
     {
                  if (y2 > y)
                                                        y += radius;
                  else
                                                        if (y2 < y)
                      y -= radius;
     }
    else
                 if (y == y2)
                   if (x2 > x)
                                                         x += radius;
                   else
```

```
                                                    if (x2 < x)
                                                      x -= radius;
    }
  }

void Drawing::FindTerminatorIntersection(Coord& x, Coord& y,
                        Coord x2, Coord y2) {
  float a00, a01, a10, scale, a20, a21;
  int radius;
  Transformer t;
  GetPictureTT(t);
  t.GetEntries(a00, a01, a10, scale, a20, a21);
  radius = (int) ((float) OperatorRadius * scale);
  double dx, dy;
  dx = (double) (x2 - x);
  dy = (double) (y2 - y);
  int steep = fabs(dy) > fabs(dx);
  if (steep) {
    if (dy < 0)
      radius = -radius;
    y += radius;
    x += (Coord) (dx * radius / dy);
  }
  else {
    if (dx < 0)
      radius = -radius;
    x += radius;
    y += (Coord) (dy * radius / dx);
  }
}

// Finds the operator that contains the given points and modifies
// those points to be where the data flow intersects with the operator

Vertex* Drawing::SetEndptsInOperator(Coord& x1, Coord& y1,
                        Coord& x2, Coord& y2) {
  Vertex* es1 = EndptsInOperator(x1,y1);
  if (es1)
    if (es1->GetClassId() == OPERATOR)
                        FindOperatorIntersection(x1,y1,x2,y2);
    else
      FindTerminatorIntersection(x1,y1,x2,y2);
  return es1;
 }

// Add ellipse to operator list

void Drawing::OperatorAppend(EllipseSelection* es, int id)
 {
  Operator* os = new Operator(es);
  os->SetId(id);
  ol->Append(new VertexNode(os));
 }

void Drawing::OperatorAppend(RectSelection* rs, int id)
 {
  Terminator* ts = new Terminator(rs);
  ts->SetId(id);
  ol->Append(new VertexNode(ts));
 }

// Add data flow to operator list

void Drawing::DataFlowSplineAppend(BSplineSelection* ss, Vertex* op1,
                        Vertex* op2)
 {
  Edge* e = new Edge(ss,op1,op2);
  el->Append(new EdgeNode(e));
  if (op1)
    op1->AddOutput(e);
  if (op2)
    op2->AddInput(e);
 }

// For all operators selected, add their mets tags, and labels to the
// selection list
```

```
void Drawing::AddTextToSelectionList()
 {
 int op_index = 0, sl_index = 0;
 int *op_index_array = new int[sl->Size()];
 for (sl ->First(); !sl->AtEnd(); sl->Next()) {
   int cid = sl->GetCur()->GetSelection()->GetClassId();
   if (cid == TERMINATOR || cid == OPERATOR)
     op_index_array[op_index++] = sl_index;
   ++sl_index;
  }
 Selection* s;
 TextSelection* ts;
 TextSelection* met_ts;
 TextSelection* tag_ts;
 for (int i = 0; i < op_index; ++i) {
   s = sl->Index(op_index_array[i])->GetSelection();
   Vertex *op = (Vertex *) s->GetOwner();
   if (op) {
     ts = op->GetLabel();
                  if (ts)
       if (!sl->Find(ts))
                     sl->Append(new SelectionNode(ts));
                  met_ts = op->GetMET();
                  if (met_ts)
                    if (!sl->Find(met_ts))
                      sl->Append(new SelectionNode(met_ts));
                  tag_ts = op->GetTag();
                  if (tag_ts)
       if (!sl->Find(tag_ts))
                     sl->Append(new SelectionNode(tag_ts));
    }
  }
}

// For all operators selected, add their associated self loops and thei
// labels to the selection list

void Drawing::AddSelfLoopsToSelectionList()
 {
  int op_index = 0, sl_index = 0;
  int *op_index_array = new int[sl->Size()];
  for (sl ->First(); !sl->AtEnd(); sl->Next()) {
    int cid = sl->GetCur()->GetSelection()->GetClassId();
                if (cid == OPERATOR || cid == TERMINATOR)
                  op_index_array[op_index++] = sl_index;
                ++sl_index;
   }
  Selection* s;
  EdgeList *sll;
  for (int i = 0; i < op_index; ++i)
   {
                s = sl->Index(op_index_array[i])->GetSelection();
                Vertex *op = (Vertex *) s->GetOwner();
                if (op) {
                  sll = op->GetOutputs();
                  for (sll->First(); !sll->AtEnd(); sll->Next()) {
        Edge *e = sll->GetCur()->GetEdge();
        if (e->IsState())
                                                    sl->Append(new SelectionNode(e->GetBSpline()));
                                                TextSelection* ts = e->GetLabel();
        if (ts && !sl->Find(ts))
                                                    sl->Append(new SelectionNode(ts));
      }
     }
   }
 }

// Modify all data flows associated with the given operator to move
// to the new location that the operator moved to

void Drawing::ReplaceAssociatedObjects(Vertex* vn, DrawingView* dv)
 {
 if (vn) {
   ReplaceInputDFSplines(vn, dv);
   ReplaceOutputDFSplines(vn, dv);
```

```
  }
}

// Modify all input data flows associated with an operator
// Changed 2/6/91
// By Patrick D. Barnes
// Description: Fixed bug where new endpoint calculation is hosed when
//  picture is zoomed, resized, or scrolled. Previous code did not
//  transform the original spline coordinates before and after calc
//  of new endpoint. "GetCenter" is in absolute Coord, but GetOriginal
//  and CreateReshapedCopy require original Coords.

void Drawing::ReplaceInputDFSplines(Vertex *op, DrawingView* dv)
 {
  float fx, fy;
  Coord cx, cy;
  op->GetShape()->GetCenter(fx, fy);
  cx = (Coord) fx;
  cy = (Coord) fy;
  Coord *x_new, *y_new;
  Coord x_new_n, y_new_n;
  Coord *x_old, *y_old;
  Coord tx, ty;
  BSplineSelection* oldss;
  BSplineSelection* newss;
  EdgeList* isl = op->GetInputs();
  int size = isl->Size();
  int index = 0;
  isl->First();
  while (index < size) {
   Edge *e = isl->GetCur()->GetEdge();
   if (e->IsStream()) {
                oldss = e->GetBSpline();
                TextSelection* ts = e->GetLabel();
    TextSelection* lat_ts = e->GetLatency();
                int num = oldss->GetOriginal(x_old, y_old);

    // Transform the original point next to the end into absolute
                // Coordinates. Requires both graphic and piture transforms.
                Transformer *t = oldss->GetTransformer();
                Transformer pt;
                GetPictureTT(pt);
                t->Transform(x_old[num-2], y_old[num-2], tx, ty);
    pt.Transform(tx,ty);

                x_new_n = cx;
                y_new_n = cy;
    if (op->GetClassId() == OPERATOR)
                  FindOperatorIntersection(x_new_n, y_new_n, tx, ty);
    else
                  FindTerminatorIntersection(x_new_n, y_new_n, tx, ty);

    // Transform the point back to original Coordinates.
                // Requires both graphic and piture transforms.
                pt.InvTransform(x_new_n, y_new_n, tx, ty);
                t->InvTransform(tx, ty, x_new_n, y_new_n);
                x_new = x_old;
                y_new = y_old;
                x_new[num-1] = x_new_n;
                y_new[num-1] = y_new_n;
                newss = (BSplineSelection*) oldss->CreateReshapedCopy(x_new,y_new,num
                                                                      ,DATAFLOW_SPLINE);
                ReplaceChange* rc = new ReplaceChange(this, dv, oldss, newss);
                rc->Do();
                ReplaceLabel(newss, ts);
                if (ts)
                  ReplaceLatency(ts, lat_ts);
    else
                  ReplaceLatency(newss, lat_ts);
   }
   ++index;
   isl->Index(index);
  }
 }

// Replace all output data flows connected to operator
```

```
// Changed 2/6/91
// By Patrick D. Barnes
// Description: Fixed bug where new endpoint calculation is hosed when
//  picture is zoomed, resized, or scrolled. Previous code did not
//  transform the original spline coordinates before and after calc
//  of new endpoint. "GetCenter" is in absolute Coord, but GetOriginal
//  and CreateReshapedCopy require original Coords.


void Drawing::ReplaceOutputDFSplines(Vertex *op, DrawingView* dv)
 {
  float fx, fy;
  Coord cx, cy;
  op->GetShape()->GetCenter(fx, fy);
  cx = (Coord) fx;
  cy = (Coord) fy;
  Coord* x_new;
  Coord* y_new;
  Coord x_new_0, y_new_0;
  Coord* x_old;
  Coord* y_old;
  Coord tx, ty;
  BSplineSelection* oldss;
  BSplineSelection* newss;

  EdgeList* osl = op->GetOutputs();
  int size = osl->Size();
  int index = 0;
  osl->First();
  while (index < size) {
   Edge *e = osl->GetCur()->GetEdge();
   if (e->IsStream()) {
                oldss = osl->GetCur()->GetEdge()->GetBSpline();
                TextSelection* ts = e->GetLabel();
    TextSelection* lat_ts = e->GetLatency();
                int num = oldss->GetOriginal(x_old, y_old);
                x_new_0 = cx;
                y_new_0 = cy;

    // Transform the original point next to the end into absolute
                // Coordinates. Requires both graphic and piture transforms.
                Transformer *t = oldss->GetTransformer();
                Transformer pt;
                GetPictureTT(pt);
                t->Transform(x_old[1], y_old[1], tx, ty);
    pt.Transform(tx,ty);

    if (op->GetClassId() == OPERATOR)
                  FindOperatorIntersection(x_new_0, y_new_0, tx, ty);
    else
                  FindTerminatorIntersection(x_new_0, y_new_0, tx, ty);

    // Transform the point back to original Coordinates.
                // Requires both graphic and piture transforms.
                pt.InvTransform(x_new_0, y_new_0, tx, ty);
                t->InvTransform(tx, ty, x_new_0, y_new_0);
                x_new = x_old;
                y_new = y_old;
                x_new[0] = x_new_0;
                y_new[0] = y_new_0;
                newss = (BSplineSelection*) oldss->CreateReshapedCopy(x_new,y_new,num
                                                                      ,DATAFLOW_SPLINE);
                ReplaceChange* rc = new ReplaceChange(this, dv, oldss, newss);
                rc->Do();
                ReplaceLabel(newss, ts);
                if (ts)
                  ReplaceLatency(ts, lat_ts);
                else
                  ReplaceLatency(newss, lat_ts);
   }
   ++index;
   osl->Index(index);
  }
}

void Drawing::ReplaceLabel(Selection* newsel, TextSelection* ts) {
```

```
  if (ts) {
                SelectionList* temp = GetSelections();
    Select(newsel);
    Extend(ts);
                Align(Center, Center);
                Select(temp);
  }
}

void Drawing::ReplaceLatency(Selection* newsel, TextSelection* lat_ts) {
  if (lat_ts) {
                SelectionList* temp = GetSelections();
    Select(newsel);
    Extend(lat_ts);
                Align(Center, Center);
                Align(Top, Bottom);
                Select(temp);
  }
}

// add Trigger condition  to operator selection list

void Drawing::TriggerAppend(TextSelection* if_ts, EllipseSelection* op) {
  ((Operator *)op->GetOwner())->AddTrigger_If(if_ts->GetString());
}

void Drawing::CommentAppend(TextSelection* ts) {
  cl->Append(new CommentNode(ts));
}

// write PSDL specification of drawing to temporary file to be edited

void Drawing::WritePSDLSpec(FILE *psdl) {
  const char *name = state->GetDrawingName();
  fprintf(psdl,"%s",OPER_TKN);
  if (name)
    fprintf(psdl,"%s\n",name);
  else
    fprintf(psdl,"%s\n",ID_TKN);
  fprintf(psdl,"%s%s%s%s",SPEC_TKN,DESC_TKN,TEXT_TKN,END_TKN);
}

// write the PSDL graph to a file

void Drawing::WritePSDLGraph(FILE* iptr)
 {
  fprintf(iptr,GR_TKN);
  WriteVertices(iptr);
  WriteEdges(iptr);
 }

// write PSDL edges to file containing PSDL graph

void Drawing::WriteEdges(FILE* fptr) {
  for (el->First(); !el->AtEnd(); el->Next()) {
    Edge *e = el->GetCur()->GetEdge();
    if (e->IsState())
      continue;
    char* froml = e->GetFromOp() ? e->GetFromVertexLabel() : EXT_TKN;
    froml = froml ? froml : ID_TKN;
    char* tol = e->GetToOp() ? e->GetToVertexLabel() : EXT_TKN;
    tol = tol ? tol : ID_TKN;
    char* label = e->GetLabel() ? e->GetValidLabelString() : ID_TKN;
    char* latency = e->GetLatencyString();
    if (latency)
                fprintf(fptr, "%s %s : %s %s -> %s\n", EDGE_TKN, label,
                    latency, froml, tol);
    else
                fprintf(fptr, "%s %s %s -> %s\n", EDGE_TKN, label,
                                                        froml, tol);
    delete [] froml;
    delete [] tol;
    delete [] label;
    delete [] latency;
  }
}
```

```
// write PSDL vertices to file containing PSDL graph

void Drawing::WriteVertices(FILE* fptr) {
  for (ol->First(); !ol->AtEnd(); ol->Next()) {
    TextSelection* ts = ol->GetCur()->GetVertex()->GetLabel();
    char *label = ts ? ts->GetValidString() : ID_TKN;
    ts = ol->GetCur()->GetVertex()->GetMET();
    char *met = ts ? ts->GetString() : 0;
    fprintf(fptr, "%s %s ", VER_TKN, label);
    if (met)
                fprintf(fptr, ": %s\n", met);
    else
                fprintf(fptr,"\n");
    delete [] met;
    delete [] label;
  }
}

void Drawing::WritePSDLConstraints(FILE *fptr) {
  int count = 0;
  for (ol->First();!ol->AtEnd();ol->Next()) {
    Vertex *op = ol->GetCur()->GetVertex();
    if (op->HasConstraints()) {
      if (!count++)
        fprintf(fptr,CON_TKN);
      op->WritePSDLConstraints(fptr);
    }
  }
}

void Drawing::WritePSDLImpl(FILE *fptr) {
  fprintf(fptr,IMP_TKN);
  WritePSDLGraph(fptr);
  el->WriteStreamTypes(fptr,STREAM_TKN);
  WritePSDLConstraints(fptr);
  fprintf(fptr, "%s", END_TKN);
}


// write psdl specification file and psdl implementation file and write
// psdl specification file for all the operators and write file needed
// to rebuild operator list when reentering editor

void Drawing::WriteDFDFiles(char* dir, const char* prototype_name)
  {
    char* spec_filename = new char[strlen(dir) + strlen(prototype_name)
                                                    + SPEC_EXT_LEN + 1];
    strcpy(spec_filename,dir);
    strcat(spec_filename,prototype_name);
    strcat(spec_filename,SPEC_PSDL_EXT);
    FILE* sptr = fopen(spec_filename, "w");
    WritePSDLSpec(sptr);
    fclose(sptr);
    delete spec_filename;

    char* imp_filename = new char[strlen(dir) + strlen(prototype_name)
                                                    + IMP_EXT_LEN + 1];
    strcpy(imp_filename,dir);
    strcat(imp_filename,prototype_name);
    strcat(imp_filename,IMP_PSDL_EXT);
    FILE* iptr = fopen(imp_filename, "w");

    WritePSDLImpl(iptr);

    fclose(iptr);
    delete [] imp_filename;

    WritePSDLForAllOperators(dir, prototype_name);

    char* graph_filename = new char [strlen(dir) + strlen(prototype_name) +
                                                    DFD_EXT_LEN + 1];
    strcpy(graph_filename,dir);
    strcat(graph_filename,prototype_name);
    strcat(graph_filename,DFD_EXT);
    FILE* gptr = fopen(graph_filename,"w");
```

```
  WriteDFDInfo(gptr);
  fclose(gptr);
  delete graph_filename;
}

// read psdl specification file and psdl implementation file and read
// psdl specification file for all the operators and read in file to
// rebuild operator list

void Drawing::ReadDFDFiles(char* dir, const char* prototype_name)
 {
  char* graph_filename = new char [strlen(dir) + strlen(prototype_name) +
                                                          DFD_EXT_LEN + 1];
  strcpy(graph_filename,dir);
  strcat(graph_filename,prototype_name);
  strcat(graph_filename,DFD_EXT);
  FILE* gptr = fopen(graph_filename,"r");
  if (gptr != nil)
   {
                ReadDFDInfo(gptr);
                fclose(gptr);
   }
  delete graph_filename;
  CheckForComposites(dir, prototype_name);
 }

// write PSDL specificatio for each operator to a file created by contenating
// the operator's identifier with the prototype name

void Drawing::WritePSDLForAllOperators(char* dir, const char* prototype_name)
 {
  for (ol->First(); !ol->AtEnd(); ol->Next())
   {
                TextSelection* ts  = ol->GetCur()->GetVertex()->GetLabel();
                if (ts)
    {
                   char* fixed_name = ts->GetValidString();
                   char* filename = new char[strlen(dir) + strlen(prototype_name)
                                                          + strlen(fixed_name) + SPEC_EXT_LEN + 2];
      strcpy(filename,dir);
                   strcat(filename,prototype_name);
                   strcat(filename,".");
                   strcat(filename,fixed_name);
                   strcat(filename,SPEC_PSDL_EXT);
                   FILE* fptr = fopen(filename,"w");
                   ol->GetCur()->GetVertex()->WritePSDL(fptr);
                   fclose(fptr);
                   delete [] fixed_name;
                   delete [] filename;
    }
   }
 }

// Add the graphics selection s to the current drawing
// Created: 10/15/94 by C.S. Eagle for .graph format change

void Drawing::AddInputSelection(Selection *s) {
  Select(s);
  Append();
  drawingview->Added();
  drawingview->EraseHandles();
}

// ReadFgColor and ReadBgColor read color index values from the file
// pointed to by fptr and returns an IColor *
// Created: 10/15/94 by C.S. Eagle for .graph format change

IColor *Drawing::ReadFgColor(FILE *fptr) {
  MapIColor* mf = state->GetMapIFgColor();
  int cnum;
  fscanf(fptr,"%d\n",&cnum);
  return mf->Index(cnum);
}

IColor *Drawing::ReadBgColor(FILE *fptr) {
  MapIColor* mf = state->GetMapIBgColor();
```

```
  int cnum;
  fscanf(fptr,"%d\n",&cnum);
  return mf->Index(cnum);
}

// WriteFgColor and WriteBgColor write color index values for IColor *c
// to file *fptr
// Created: 10/15/94 by C.S. Eagle for .graph format change

void Drawing::WriteFgColor(FILE *fptr, IColor *c) {
  int cnum = state->GetFgColorIndex(c);
  fprintf(fptr,"%d\n",cnum);
}

void Drawing::WriteBgColor(FILE *fptr, IColor *c) {
  int cnum = state->GetBgColorIndex(c);
  fprintf(fptr,"%d\n",cnum);
}

// ReadFont creates a new IFont from information in file *fptr
// Created: 10/15/94 by C.S. Eagle for .graph format change

IFont *Drawing::ReadFont(FILE *fptr) {
  MapIFont* mf = state->GetMapIFont();
  int fontnum;
  fscanf(fptr,"%d\n",&fontnum);
  return mf->Index(fontnum);
}

// WriteFont save font information to file *fptr
// Created: 10/15/94 by C.S. Eagle for .graph format change

void Drawing::WriteFont(FILE *fptr, IFont *f) {
  int fontnum = state->GetFontIndex(f);
  fprintf(fptr,"%d\n",fontnum);
}

// read a point rom a file
// Created: 10/17/94 by C.S. Eagle for .graph format change

void Drawing::ReadPoint(FILE *fptr,int textflag, Coord &x, Coord &y) {
  fscanf(fptr, "%d %d\n", &x, &y);
  if (!textflag) {
    Transformer *t = state->GetGraphicGS()->GetTransformer();
    t->InvTransform(x,y);
   }
}

// write a point to a file
// Created: 10/17/94 by C.S. Eagle for .graph format change

void Drawing::WritePoint(FILE *fptr, Transformer *t, Coord x, Coord y) {
  t->Transform(x,y);
  fprintf(fptr, "%d %d\n",x,y);
}

// WriteTS writes a TextSelection to file *fptr
// Created: 10/15/94 by C.S. Eagle for .graph format change

void Drawing::WriteTS(TextSelection *ts, FILE *fptr) {
  if (!ts) {
    fprintf(fptr,"0\n");
    return;
  }
  char *text = ts->GetString();
  if (!text) {
    fprintf(fptr,"0\n");
    return;
  }
  fprintf(fptr, "%d\n%s\n",strlen(text),text);
  WriteFont(fptr, (IFont *) ts->GetFont());
  WriteFgColor(fptr, (IColor *) ts->GetFgColor());
  fprintf(fptr,"0\n");
  WritePoint(fptr,ts->GetTransformer(),0,0);
}
```

```
// ReadTS reads a TextSelection from file *fptr
// Created: 10/15/94 by C.S. Eagle for .graph format change

TextSelection *Drawing::ReadTS(ClassId cid, FILE *fptr) {
  TextSelection *rval = 0;
  char text[256];
  int len;
  char format[15];
  fscanf(fptr,"%d\n",&len);
  if (len) {
    sprintf(format, "%%%dc\n",len);
    fscanf(fptr, format, text);
    text[len] = 0;
    rval = new TextSelection(cid,text,len,state->GetTextGS());
    rval->SetFont(ReadFont(fptr));
    rval->SetColors(ReadFgColor(fptr),new IColor("White"));
    int defPos;
    fscanf(fptr,"%d\n",&defPos);
    if (!defPos) {
      rval->SetDefPosition(false);
      Coord x, y;
      ReadPoint(fptr,1,x,y);
      rval->SetTransformer(new Transformer(1,0,0,1,x,y));
    }
    AddInputSelection(rval);
  }
  return rval;
}

// WriteGraphEdge saves all the information needed to
// represent an edge in the graph
// Created: 10/15/94 by C.S. Eagle for .graph format change

void Drawing::WriteGraphEdge(FILE *outfile, Edge *df) {
  fprintf(outfile, "%u\n", DATAFLOW_SPLINE);
  int to_index;
  if (df->GetToOp())
    to_index = df->GetToOp()->GetId();
  else
    to_index = -1;
  int from_index;
  if (df->GetFromOp())
    from_index = df->GetFromOp()->GetId();
  else
    from_index = -1;

  BSplineSelection* flow = df->GetBSpline();
  Coord *x, *y;
  int npts = flow->GetOriginal(x,y);
  fprintf(outfile,"%d %d %d %u\n", npts,
        from_index, to_index, df->StreamType());
  for(int i = 0; i<npts; i++)
    WritePoint(outfile,flow->GetTransformer(),x[i],y[i]);

  WriteFgColor(outfile,(IColor *) flow->GetFgColor());
  WriteTS(df->GetLabel(), outfile);
  if (from_index != to_index)
    WriteTS(df->GetLatency(), outfile);
}

//  Writes the operator's attributes to disk.
// Created: 10/15/94 by C.S. Eagle for .graph format change

void Drawing::WriteGraphVertex(FILE *outfile, Vertex *os) {
  int cid = os->GetClassId();
  fprintf(outfile, "%u\n", cid);
  fprintf(outfile,"%d\n", os->GetId());
  Selection *s = os->GetShape();
  if (cid == OPERATOR) {
    int x0, y0, rx, ry;
    EllipseSelection *es = (EllipseSelection *) s;
    es->GetOriginal(x0,y0,rx,ry);
    WritePoint(outfile, es->GetTransformer(),x0,y0);
    fprintf(outfile, "%d %d\n",rx,ry);
  }
  else {
```

```
    Coord l, b, r, t;
    RectSelection *rs = (RectSelection *) s;
    rs->GetOriginal2(l,b,r,t);
    WritePoint(outfile, rs->GetTransformer(),l,b);
    WritePoint(outfile, rs->GetTransformer(),r,t);
  }
  WriteFgColor(outfile,(IColor *) s->GetFgColor());
  WriteBgColor(outfile,(IColor *) s->GetBgColor());
  WriteTS(os->GetLabel(),outfile);
  if (cid == OPERATOR)
    WriteTS(os->GetMET(),outfile);
  else
    fprintf(outfile,"0\n");
}

// write information to file to be able to rebuild operator list when
// coming back into editor
// Created: 10/15/94 by C.S. Eagle for .graph format change

void Drawing::WriteDFDInfo(FILE* fptr) {

  for (ol->First(); !ol->AtEnd(); ol->Next())
    WriteGraphVertex(fptr,ol->GetCur()->GetVertex());

  for (el->First(); !el->AtEnd(); el->Next())
    WriteGraphEdge(fptr,el->GetCur()->GetEdge());

  for (cl->First(); !cl->AtEnd(); cl->Next()) {
    fprintf(fptr,"%u\n",COMMENT);
    WriteTS(cl->GetCur()->GetSelection(), fptr);
  }

  fprintf(fptr, "%u\n", END_MARKER);

  fprintf(fptr,"%d %d\n",idraw->wWidth(),idraw->wHeight());

  Perspective *p = drawingview->GetPerspective();
  fprintf(fptr,"%d %d %d %d\n",p->curx, p->cury, p->curwidth, p->curheight);

  WriteFgColor(fptr, state->GetFgColor());
  WriteBgColor(fptr, state->GetBgColor());
  WriteFont(fptr, state->GetFont());
}

// Builds the stream from disk.
// Created: 10/15/94 by C.S. Eagle for .graph format change

void Drawing::ReadGraphEdge(FILE *infile) {
  Coord *x, *y;
  int is_stream;
  int from_index, to_index;
  int npts;
  fscanf(infile, "%d %d %d %u\n", &npts,
      &from_index, &to_index, &is_stream);
  x = new Coord[npts];
  y = new Coord[npts];
  for(int i = 0; i<npts; i++)
    ReadPoint(infile,0,x[i],y[i]);

  IBrush *oldBrush = state->GetBrush();
  MapIBrush *mb = state->GetMapIBrush();

  IPattern *temp = state->GetPattern();
  state->SetPattern(state->GetArrowPattern());
  state->SetFgColor(ReadFgColor(infile));

  BSplineSelection *flow;
  if (from_index == to_index) {
    state->SetBrush(mb->FindOrAppend(0,0xffff,1,0,1));
    flow = new
      BSplineSelection(SELFLOOP,x,y,npts,state->GetGraphicGS());
  }
  else {
    if (is_stream == 2)
      state->SetBrush(mb->FindOrAppend(0,0xf0f0,2,0,1));
    flow = new
```

```
    BSplineSelection(DATAFLOW_SPLINE,x,y,npts,state->GetGraphicGS());
  }

  DataFlowSplineAppend(flow,ol->FindOp(from_index),ol->FindOp(to_index));

  state->SetBrush(oldBrush);
  state->SetPattern(temp);
  AddInputSelection(flow);
  TextSelection *name;
  if (from_index == to_index)
    name = ReadTS(LABEL_SL,infile);
  else
    name = ReadTS(LABEL_DF,infile);
  Edge *e = (Edge *) flow->GetOwner();
  if (name) {
    e->SetLabel(name);
    if (name->GetDefPosition())
      flow->Align(Center, name, Center);
  }
  if (from_index != to_index) {
    TextSelection *lat = ReadTS(LAT_DF,infile);
    if (lat) {
      e->SetLatency(lat);
      if (lat->GetDefPosition()) {
        if (name) {
          name->Align(Center,lat,Center);
          name->Align(Bottom,lat,Top);
        }
        else
          flow->Align(Center, lat, Center);
      }
    }
  }
}

// Builds an operator from a disk file.
// Created: 10/15/94 by C.S. Eagle for .graph format change

void Drawing::ReadGraphVertex(FILE *infile,int cid) {
  int id;
  fscanf(infile,"%d\n",&id);
  int x0, y0, rx, ry;
  ReadPoint(infile,0,x0,y0);
  if (cid == OPERATOR)
    fscanf(infile, "%d %d\n", &rx, &ry);
  else
    ReadPoint(infile,0,rx,ry);
  state->SetFgColor(ReadFgColor(infile));
  state->SetBgColor(ReadBgColor(infile));

  Selection *s;
  if (cid == OPERATOR) {
    EllipseSelection *es = new
      EllipseSelection(x0, y0, rx, ry,state->GetGraphicGS());
    OperatorAppend(es,id);
    s = es;
  }
  else {
    RectSelection *rs = new
      RectSelection(x0, y0, rx, ry,state->GetGraphicGS());
    OperatorAppend(rs,id);
    s = rs;
  }

  AddInputSelection(s);
  TextSelection *name = ReadTS(LABEL_OP, infile);
  TextSelection *mets = ReadTS(MET_OP, infile);
  Vertex *op = (Vertex *) s->GetOwner();
  if (name) {
    op->SetLabel(name);
    if (name->GetDefPosition())
      s->Align(Center, name, Center);
  }
  if (mets) {
    ((Operator *)op)->SetMET(mets);
    if (mets->GetDefPosition()) {
```

```
      s->Align(Center, mets, Center);
      s->Align(Top, mets, Bottom);
    }
  }
}

// Read in file to rebuild the graphics picture, and the underlying
// DFD structure
// Created: 10/15/94 by C.S. Eagle for .graph format change

void Drawing::ReadDFDInfo(FILE* fptr) {
  ClassId cid;
  TextSelection *ts;
  state->SetTextGS(0,0,state->GetTextPainter());
  do {
   fscanf(fptr, "%u", &cid);
   switch (cid) {
     case TERMINATOR:
     case OPERATOR:
       ReadGraphVertex(fptr,cid);
                   break;
     case DATAFLOW_SPLINE:
       ReadGraphEdge(fptr);
                   break;
     case COMMENT:
       ts = ReadTS(COMMENT,fptr);
       cl->Append(new CommentNode(ts));
       break;
    }
  } while (cid != END_MARKER);

  Shape *s = new Shape;
  int x, y, w, h;
  fscanf(fptr,"%d %d\n",&w,&h);
  s->Rect(w,h);
  idraw->Reshape(*s);

  Perspective p;
  fscanf(fptr,"%d %d %d %d\n",&x, &y, &w,&h);
  p.Init(x,y,w,h);
  drawingview->Adjust(p);

  state->SetFgColor(ReadFgColor(fptr));
  state->SetBgColor(ReadBgColor(fptr));
  state->SetFont(ReadFont(fptr));
  Clear();
  drawingview->EraseHandles();
}

void Drawing::GenerateAda(FILE *fptr, Vertex *v) {
  v->GenerateAda(fptr);
}

void Drawing::TagOperator(Vertex *v, char *tag) {
  TextSelection *tagsel =
      new TextSelection(tag,strlen(tag),state->GetTextGS());
  AddInputSelection(tagsel);
  v->SetTag(tagsel);
  TextSelection *ls = v->GetLabel();
  ls->Align(Center, tagsel, Center);
  ls->Align(Top, tagsel, Bottom);
  drawingview->EraseHandles();
  Clear();
}

void Drawing::CheckForComposites(char* dir, const char* prototype_name)
{
  for (ol->First(); !ol->AtEnd(); ol->Next())
    {
                TextSelection* ts = ol->GetCur()->GetVertex()->GetLabel();
                if (ts)
      {
                  char *fixed_name = ts->GetValidString();
                  char* filename = new char[strlen(dir) + strlen(prototype_name)
                                                        + strlen(fixed_name) + DFD_EXT_LEN + 2];
       strcpy(filename,dir);
```

```
        strcat(filename,prototype_name);
                        strcat(filename,".");
                        strcat(filename,fixed_name);
                        strcat(filename,DFD_EXT);
        if (Exists(filename))
          TagOperator(ol->GetCur()->GetVertex(),"*");
                        delete [] fixed_name;
                        delete [] filename;
      }
    }
  }
```

drawingview.h

```
#ifndef drawingview_h
#define drawingview_h

#include <InterViews/Graphic/grblock.h>

// Declare imported types.

class Damage;
class Page;
class Rubberband;
class Selection;
class SelectionList;
class State;
class TextEdit;

// A DrawingView displays the user's drawing.

class DrawingView : public GraphicBlock {
public:

  DrawingView(Page*);
  ~DrawingView();

  void SetSelectionList(SelectionList*);
  void SetState(State*);
  void SetTools(Interactor*);

  void Draw();
  void Handle(Event&);

  void Manipulate(
                Event&, Rubberband*, int, boolean constrain=true, boolean erase=true
  );
  void Edit(Event&, TextEdit*, Graphic* = nil);

  void DrawHandles();
  void RedrawHandles();
  void EraseHandles();
  void EraseExcessHandles(SelectionList*);
  void ErasePickedHandles(Selection*);
  void ErasePickedHandles(SelectionList*);
  void EraseUngraspedHandles(Selection*);
  void ResetAllHandles();

  void Added();
  void Damaged();
  void Repair();

  void Magnify(Coord, Coord, Coord, Coord);
  void Reduce();
  void Enlarge();
  void NormalSize();
  void ReduceToFit();
  void CenterPage();

protected:

  void Reconfig();
  void Redraw(Coord, Coord, Coord, Coord);
  void Resize();
  float LimitMagnification(float);
```

```
    void StartTextEditing(Event&, TextEdit*, Graphic*);
    void EndTextEditing();
    void RedrawTextEditor();

    Damage* damage;                         // keeps track of damaged areas of drawing
    Graphic* gs;                            // remembers text being edited
    Page* page;                             // snaps points to grid
    Painter* rasterxor;                     // stores painter with which to draw handles
    SelectionList* sl;                      // lists current Selections
    State* state;                           // stores Graphic and nonGraphic attributes
    TextEdit* textedit;                     // redraws in-place texteditor if necessary
    Interactor* tools;                      // handles events

};

#endif
```

drawingview.c

```
#include "drawingview.h"
#include "istring.h"
#include "listselectn.h"
#include "page.h"
#include "slpict.h"
#include "state.h"
#include "textedit.h"
#include <InterViews/Graphic/damage.h>
#include <InterViews/event.h>
#include <InterViews/painter.h>
#include <InterViews/perspective.h>
#include <InterViews/rubband.h>
#include <InterViews/sensor.h>
#include <InterViews/shape.h>
#include <stdlib.h>
#define _POSIX_SOURCE
#include <math.h>

// DrawingView caches its canvas' contents if possible to speed up
// redrawing after expose events.

static const int PAD = 0;// we don't want any padding around graphic

DrawingView::DrawingView (Page* p) : (updownEvents, p, PAD, Center, Binary) {
    damage = nil;
    gs = nil;
    page = p;
    rasterxor = nil;
    sl = nil;
    state = nil;
    textedit = nil;
    tools = nil;
    SetCanvasType(CanvasSaveContents);
}

// Free storage allocated to store members.

DrawingView::~DrawingView () {
    delete damage;
    Unref(rasterxor);
}

// Define access functions to set members' values.  Only Idraw sets
// their values.

void DrawingView::SetSelectionList (SelectionList* slist) {
    sl = slist;
}

void DrawingView::SetState (State* s) {
    state = s;
}

void DrawingView::SetTools (Interactor* t) {
    tools = t;
```

```
}

// Draw draws the entire view.  Draw calls Check for its side effect
// of flushing any redraws caused by a dialog box's removal before
// drawing the view.

void DrawingView::Draw () {
  if (graphic != nil) {
                Graphic* backup = graphic;
                graphic = nil;
                Check();
                graphic = backup;

                GraphicBlock::Draw();
                damage->Reset();
                ResetAllHandles();
                RedrawHandles();
                RedrawTextEditor();
  }
}

// Handle delegates input events to the tools.

void DrawingView::Handle (Event& e) {
  tools->Handle(e);
}

// Manipulate lets the user manipulate the Rubberband with the mouse
// until a specified event occurs.

void DrawingView::Manipulate (Event& e, Rubberband* rubberband, int et,
boolean constrain, boolean erase) {
  rubberband->SetPainter(rasterxor);
  rubberband->SetCanvas(canvas);
  Listen(allEvents);
  while (e.eventType != et) {
                if (e.eventType == MotionEvent) {
                  rubberband->Track(e.x, e.y);
                }
                Read(e);
                if (constrain) {
                  page->Constrain(e.x, e.y);
                }
  }
  Listen(input);
  if (erase) {
                rubberband->Erase();
  }
}

// Edit lets the user enter text into the drawing.

void DrawingView::Edit (Event& e, TextEdit* textedit, Graphic* gs) {
  StartTextEditing(e, textedit, gs);

  textedit->Grasp(e);
  Listen(allEvents);
  while (textedit->Editing(e)) {
                Read(e);
  }
  UnRead(e);
  Listen(input);

  EndTextEditing();
}

// DrawHandles tells all the Selections to draw their handles unless
// they've already been drawn.

void DrawingView::DrawHandles () {
  for (sl->First(); !sl->AtEnd(); sl->Next()) {
                sl->GetCur()->GetSelection()->DrawHandles(rasterxor, canvas);
  }
}

// RedrawHandles tells all the Selections to redraw their handles
```

```
// whether or not they've already been drawn.

void DrawingView::RedrawHandles () {
  for (sl->First(); !sl->AtEnd(); sl->Next()) {
                sl->GetCur()->GetSelection()->RedrawHandles(rasterxor, canvas);
  }
}

// EraseHandles tells all the Selections to erase their handles unless
// they've already been erased.

void DrawingView::EraseHandles () {
  for (sl->First(); !sl->AtEnd(); sl->Next()) {
                sl->GetCur()->GetSelection()->EraseHandles(rasterxor, canvas);
  }
}

// EraseExcessHandles erases the excess Selections' handles if it
// doesn't find the Selections in the current SelectionList.

void DrawingView::EraseExcessHandles (SelectionList* newsl) {
  for (sl->First(); !sl->AtEnd(); sl->Next()) {
                Selection* oldselection = sl->GetCur()->GetSelection();
                if (!newsl->Find(oldselection)) {
                  oldselection->EraseHandles(rasterxor, canvas);
                }
  }
}

// ErasePickedHandles erases the picked Selection's handles if it
// finds the picked Selection in the SelectionList.

void DrawingView::ErasePickedHandles (Selection* pick) {
  if (sl->Find(pick)) {
                sl->GetCur()->GetSelection()->EraseHandles(rasterxor, canvas);
  }
}

// ErasePickedHandles erases the picked Selections' handles if it
// finds the picked Selections in the SelectionList.

void DrawingView::ErasePickedHandles (SelectionList* pl) {
  for (pl->First(); !pl->AtEnd(); pl->Next()) {
                Selection* pick = pl->GetCur()->GetSelection();
                if (sl->Find(pick)) {
                  sl->GetCur()->GetSelection()->EraseHandles(rasterxor, canvas);
                }
  }
}

// EraseUngraspedHandles erases all of the handles only if the
// SelectionList does not already include the picked Selection.

void DrawingView::EraseUngraspedHandles (Selection* pick) {
  if (!sl->Find(pick)) {
                EraseHandles();
  }
}

// ResetAllHandles resets all of the handles because the Selections
// may have moved out from under their handles.

void DrawingView::ResetAllHandles () {
  PictSelection* picture = page->GetPicture();
  for (picture->First(); !picture->AtEnd(); picture->Next()) {
                Selection* s = picture->GetCurrent();
                s->ResetHandles();
  }
}

// Added adds the Selections to the list of Selections in the drawing
// to be drawn for the first time.

void DrawingView::Added () {
  for (sl->First(); !sl->AtEnd(); sl->Next()) {
    damage->Added(sl->GetCur()->GetSelection());
```

```
    }
}

// Damaged adds the areas covered by the selected Selections
// (including their handles, too) to the list of damaged areas in the
// drawing to be repaired.

void DrawingView::Damaged () {
    BoxObj box;
    for (sl->First(); !sl->AtEnd(); sl->Next()) {
                    sl->GetCur()->GetSelection()->GetPaddedBox(box);
        damage->Incur(box);
    }
}

// Repair repairs the drawing's damaged areas and then redraws the
// Selections' handles. The damaged areas must have included all the
// handles for RedrawHandles to work correctly.

void DrawingView::Repair () {
    if (damage->Incurred()) {
                    damage->Repair();
                    RedrawHandles();
    }
}

// Magnify magnifies the given area to fill the view.

void DrawingView::Magnify (Coord l, Coord b, Coord r, Coord t) {
    Perspective np;
    np = *GetPerspective();
    np.curx += min(l, r);
    np.cury += min(b, t);
    np.curwidth = max(abs(r - l), 1);
    np.curheight = max(abs(t - b), 1);
    Adjust(np);
}

// Reduce reduces the drawing's magnification by a factor of two.

void DrawingView::Reduce () {
    SetMagnification(GetMagnification() / 2);
}

// Enlarge enlarges the drawing's magnification by a factor of two.

void DrawingView::Enlarge () {
    SetMagnification(2 * GetMagnification());
}

// NormalSize resets the drawing's magnification.

void DrawingView::NormalSize () {
    SetMagnification(1.);
}

// ReduceToFit reduces the drawing's magnification enough to fit all
// of the drawing in the window.

void DrawingView::ReduceToFit () {
    Perspective np;
    np = *GetPerspective();
    np.curx = np.x0;
    np.cury = np.y0;
    np.curwidth = np.width;
    np.curheight = np.height;
    Adjust(np);
}

// CenterPage recenters the drawing over the window's center.

void DrawingView::CenterPage () {
    register Perspective* p = perspective;
    Coord left, bottom, right, top;

    page->Center(mag, xmax/2, ymax/2);
```

```
    GetGraphicBox(left, bottom, right, top);
    x0 = left;
    y0 = bottom;
    p->width = right - left + 2*pad;
    p->height = top - bottom + 2*pad;
    p->curwidth = xmax + 1;
    p->curheight = ymax + 1;
    p->curx = (p->width - p->curwidth)/2;
    p->cury = (p->height - p->curheight)/2;
    p->Update();
    Draw();
}

// Reconfig gives output the page's background color, creates a
// painter for drawing the rubberbands, and asks for the smallest
// possible canvas if the user wants a small window.

void DrawingView::Reconfig () {
    Color* bg = page->GetBackgroundColor();
    if (output->GetBgColor() != bg) {
                Painter* copy = new Painter(output);
                copy->Reference();
                Unref(output);
                output = copy;
                output->SetColors(output->GetFgColor(), bg);
    }
    if (rasterxor == nil) {
                rasterxor = new Painter(output);
                rasterxor->Reference();
    }
    if (strcmp(GetAttribute("small"), "true") == 0) {
                shape->width = 0;
                shape->height = 0;
    }
    GraphicBlock::Reconfig();
}

// Redraw draws a rectangular subpart of the view.

void DrawingView::Redraw (Coord l, Coord b, Coord r, Coord t) {
    if (graphic != nil) {
                GraphicBlock::Redraw(l, b, r, t);
                rasterxor->Clip(canvas, l, b, r, t);
                for (sl->First(); !sl->AtEnd(); sl->Next()) {
                    Selection* selection = sl->GetCur()->GetSelection();
                    selection->RedrawUnclippedHandles(rasterxor, canvas);
                }
                rasterxor->NoClip();
                RedrawTextEditor();
    }
}

// Resize recreates damage in case canvas changed its value.

void DrawingView::Resize () {
    GraphicBlock::Resize();
    delete damage;
    damage = new Damage(canvas, output, graphic);
    ResetAllHandles();
}

// LimitMagnification limits the factor by which DrawingView may scale
// the view of the drawing to avoid torturing the X server. In
// addition, LimitMagnification updates State's stored magnification.
// Alternatively, State could have attached itself to DrawingView's
// perspective if it was an Interactor like MagnifView.

float DrawingView::LimitMagnification (float desired) {
    const float MIN = 1./8.;
    const float MAX = 16.;
    if (desired < MIN) {
                desired = MIN;
    } else if (desired > MAX) {
                desired = MAX;
    }
```

```
    state->SetMagnif(desired);
    return desired;
}

// StartTextEditing stores the textedit and draws it on the canvas.

void DrawingView::StartTextEditing (
    Event& e, TextEdit* textedit, Graphic* gs
) {
    DrawingView::textedit = textedit;
    DrawingView::gs = gs;

    page->Constrain(e.x, e.y);
    if (gs != nil) {
                state->SetTextGS(gs, output);
    } else {
                state->SetTextGS(e.x, e.y, output);
    }
    textedit->Redraw(
                state->GetTextPainter(), canvas, state->GetLineHt(), false
    );
}

// EndTextEditing marks the area damaged by the textedit for later
// repair and forgets it has a textedit.

void DrawingView::EndTextEditing () {
    Coord xmin, ymin, xmax, ymax;
    textedit->Bounds(xmin, ymin, xmax, ymax);
    damage->Incur(xmin, ymin, xmax, ymax);
    gs = nil;
    textedit = nil;
}

// RedrawTextEdit redraws the textedit on the canvas after a Resize or
// other asynchronous window event.

void DrawingView::RedrawTextEditor () {
    if (textedit != nil) {
                if (gs != nil) {
                    state->SetTextGS(gs, output);
                }
                textedit->Redraw(
                    state->GetTextPainter(), canvas, state->GetLineHt(), true
                );
    }
}
```

editor.h

```
#ifndef editor_h
#define editor_h

#include <InterViews/defs.h>
#include <InterViews/Std/stream.h>

// Declare imported types.

class ChangeNode;
class Chooser;
class Confirmer;
class Drawing;
class DrawingView;
class Event;
class Selecter;
class History;
class IBrush;
class IColor;
class IFont;
class IPattern;
class Interactor;
class Messager;
class Namer;
class Painter;
class RubberEllipse;
```

```
class RubberLine;
class RubberRect;
class State;
class Tools;

// An Editor lets the user perform a drawing or editing operation on
// a Drawing.

class Editor {
public:

  Editor(Interactor*);
  ~Editor();

  void SetDrawing(Drawing*);
  void SetDrawingView(DrawingView*);
  void SetState(State*);
  void SetTools(Tools*);
  void SetDirectory(char*, char*, char*);

  void HandleSelect(Event&);
  void HandleMove(Event&);
  void HandleScale(Event&);
  void HandleModify(Event&);
  void HandleMagnify(Event&);
  void HandleDecompose(Event&);
  void HandleText(Event&);
  void HandleLabel(Event&);
  void HandleMET(Event&, char * units);

  void HandleIf (Event&);

  void HandleLatency(Event& , char * units);
  void HandleBSpline(Event&);
  void HandleEllipse(Event&);
  void HandleRect(Event&);

/* ***** Start of Commented Out Code *****
  void HandleStretch(Event&);
  void HandleRotate(Event&);
  void HandleMultiLine(Event&);
  void HandlePolygon(Event&);
  void HandleClosedBSpline(Event&);
  ***** End of Commented Out Code ***** */

  void New();
  void Revert();
  void Open(const char*);
  void Open();
  void Save();
  void Commit();
  void SaveAs();
  void Print();
  void Quit(Event&);
  void Checkpoint();

  void Undo();
  void Redo();
  void Cut();
  void Copy();
  void Paste();
  void Duplicate();
  void Delete();
  void SelectAll();

   void SetBrush(IBrush*);
  void SetFgColor(IColor*);
  void SetBgColor(IColor*);
  void SetFont(IFont*);
  void SetPattern(IPattern*);

  void AlignLeftSides();
  void AlignRightSides();
  void AlignBottoms();
  void AlignTops();
  void AlignVertCenters();
```

```
    void AlignHorizCenters();
    void AlignCenters();
    void AlignLeftToRight();
    void AlignRightToLeft();
    void AlignBottomToTop();
    void AlignTopToBottom();
    void AlignToGrid();

    void Reduce();
    void Enlarge();
    void ReduceToFit();
    void NormalSize();
    void CenterPage();
    void RedrawPage();
    void GriddingOnOff();
    void GridVisibleInvisible();
    void GridSpacing();
    void Orientation();
    void ShowVersion();
    void ResetMessage(const char*);

    void ToolSet(char);

protected:

    const char* MakeFilename(const char*);
    const char* MakeSpecFilename(const char*);
    void Do(ChangeNode*);
    void InputVertices(Event&, Coord*&, Coord*&, int&);
    RubberLine* NewRubberLineOrAxis(Event&);
    RubberEllipse* NewRubberEllipseOrCircle(Event&);
    RubberRect* NewRubberRectOrSquare(Event&);
    boolean OfferToSave();
    void Reset(const char*);

    History* history;                                   // carries out and logs changes made to drawing
    Messager* numberofdialog; // displays how many graphics the drawing has
    Selecter* opendialog;                               // prompts for name of a drawing to open
    Confirmer* overwritedialog;// confirms whether to overwrite a file
    Namer* precmovedialog;    // prompts for X and Y movement in points
    Namer* precrotdialog;     // prompts for rotation in degrees
    Namer* precscaledialog;   // prompts for X and Y scaling
    Namer* printdialog;                                 // prompts for print command
    Messager* readonlydialog; // tells user drawing is readonly
    Confirmer* revertdialog; // confirms whether to revert from a file
    Selecter* saveasdialog;   // prompts for name to save drawing as
    Confirmer* savecurdialog; // confirms whether to save current drawing
    Namer* spacingdialog;     // prompts for grid spacing in points
    Messager* versiondialog; // displays idraw's version level and author
    Chooser* decomposedialog; // displays decomposition choices
    Messager* nolabeldialog;   // tells user that cannot decompose because
                                                        // operator chosen has no label

    Drawing* drawing;                                   // performs operations on drawing
    DrawingView* drawingview; // displays drawing
    State* state;                                       // stores Graphic and nonGraphic attributes
    char* dir;              // directory where prototypes are stored
    char* ddbCmd;           // name of program to update the design_database
    char* design_database;     // name of the design database to load.
    Interactor* inter;        // store the interactor (idraw) for editor
                                                        // in order to change cursor for Annotate

    Tools *tools;

};

#endif
```

editor.c

```
#include "dfdclasses.h"
#include "dfd_defs.h"
#include "dialogbox.h"
#include "drawing.h"
#include "drawingview.h"
#include "editor.h"
#include "history.h"
```

```
#include "idraw.h"
#include "istring.h"
#include "listchange.h"
#include "listselectn.h"
#include "mapipaint.h"
#include "psdllists.h"
#include "psdlcomp.h"
#include "rubbands.h"
#include "selection.h"
#include "slellipses.h"
#include "sllines.h"
#include "slpolygons.h"
#include "slsplines.h"
#include "sltext.h"
#include "state.h"
#include "textedit.h"
#include "tools.h"
#include "version.h"
#include "selecter.h"
#define _POSIX_SOURCE
#include <InterViews/Std/math.h>
#include <InterViews/cursor.h>
#include <InterViews/event.h>
#include <InterViews/transformer.h>
#include <InterViews/Graphic/util.h>
#include <sys/param.h>
#include <bstring.h>
#include <InterViews/Std/stdio.h>
#include <string.h>

extern "C" {
  int fork();
  int execlp(char*, ... );
  int exit(int);
  int wait(long*);
  int system(const char*);
}

// used to find all prototypes in directory defined in "selecter.c"

void find_prototype_names(char**, char*, const char*);

// Editor creates its history and dialog boxes.

Editor::Editor (Interactor* i) {
  history = new History(i);
  numberofdialog = new Messager(i);
  opendialog = new Selecter(i, "Select prototype to edit:", Center);
  overwritedialog= new Confirmer(i, "already exists; overwrite?");
  precmovedialog =new Namer(i,"Enter X and Y movement in printer's points:");
  precrotdialog = new Namer(i, "Enter rotation in degrees:");
  precscaledialog = new Namer(i, "Enter X and Y scaling:");
  printdialog = new Namer(i, "Enter print command:");
  readonlydialog = new Messager(i, "Drawing is readonly.");
  revertdialog = new Confirmer(i, "Really revert to original?");
  saveasdialog = new Selecter(i, "Select prototype to save:", Center);
  savecurdialog = new Confirmer(i, "Save current drawing?");
  spacingdialog = new Namer(i, "Enter grid spacing in printer's points:");
  versiondialog = new Messager(i, version);
  decomposedialog = new Chooser(i, "Choose decomposition type:",
              "Graphic Editor", "   Ada   ",
              "  Search  ");
  nolabeldialog =
      new Messager(i, "Operator has no label, cannot decompose.");

  drawing = nil;
  drawingview = nil;
  state = nil;
  inter = i;
}

// ~Editor frees storage allocated for its history and dialog boxes.

Editor::~Editor () {
  delete history;
  delete numberofdialog;
```

```
 delete opendialog;
 delete overwritedialog;
 delete precmovedialog;
 delete precrotdialog;
 delete precscaledialog;
 delete printdialog;
 delete readonlydialog;
 delete revertdialog;
 delete saveasdialog;
 delete savecurdialog;
 delete spacingdialog;
 delete versiondialog;
}

// Define access functions to set members' values.  Only Idraw sets
// their values.

void Editor::SetDrawing (Drawing* d) {
 drawing = d;
}

void Editor::SetDrawingView (DrawingView* dv) {
 drawingview = dv;
}

void Editor::SetState (State* s) {
 state = s;
}

void Editor::SetTools (Tools* i) {
 tools = i;
}

// set the directory of prototytpes to the given character string
// Changed 1/31/91
// By Patrick D. Barnes
// Description: Sets prototype directory, design_database, and dbCmd
void Editor::SetDirectory(char* d, char* db, char* c) {
 dir = new char [MAXPATHLEN + 1];
 design_database = new char[strlen(db)];
 ddbCmd = new char[strlen(c)];
 strcpy(dir,d);
 strcat(dir, "/");
 strcpy(design_database,db);
 strcpy(ddbCmd,c);
}

// HandleSelect lets the user pick a Selection if one's under the
// mouse, otherwise it lets the user manipulate a rubber rectangle to
// enclose the Selection he wants to pick.  HandleSelect clears all
// previous Selections unless the user holds down the shift key to
// extend the Selections being made.

void Editor::HandleSelect (Event& e) {
 Selection* pick = drawing->PickSelectionIntersecting(e.x, e.y);
 if (!e.shift) {  // replacing previous Selections
  drawingview->EraseHandles();
  if (pick != nil) {
   drawing->Select(pick);
  } else {
   RubberRect* rubberrect =
   new RubberRect(nil, nil, e.x, e.y, e.x, e.y);
   drawingview->Manipulate(e, rubberrect, UpEvent, false);
   Coord l, b, r, t;
   rubberrect->GetCurrent(l, b, r, t);
   delete rubberrect;

   SelectionList* picklist= drawing->PickSelectionsWithin(l, b, r, t);
   drawing->Select(picklist);
   delete picklist;
  }
 } else {    // extending Selections
  if (pick != nil) {
   drawingview->ErasePickedHandles(pick);
   drawing->Extend(pick);
  } else {
```

```
    RubberRect* rubberrect =
    new RubberRect(nil, nil, e.x, e.y, e.x, e.y);
    drawingview->Manipulate(e, rubberrect, UpEvent, false);
    Coord l, b, r, t;
    rubberrect->GetCurrent(l, b, r, t);
    delete rubberrect;

    SelectionList* picklist= drawing->PickSelectionsWithin(l, b, r, t);
    drawingview->ErasePickedHandles(picklist);
    drawing->Extend(picklist);
    delete picklist;
   }
  }
  drawingview->DrawHandles();
}

// HandleMove lets the user manipulate a sliding rectangle enclosing
// the Selections and moves them the same way when the user releases
// the button.

void Editor::HandleMove (Event& e) {
  Selection* pick = drawing->PickSelectionIntersecting(e.x, e.y);
  if (pick != nil) {
    drawingview->EraseUngraspedHandles(pick);
    drawing->Grasp(pick);
    drawingview->DrawHandles();
    Coord l, b, r, t;
    drawing->GetBox(l, b, r, t);
    state->Constrain(e.x, e.y);
    SlidingRect* slidingrect =
      new SlidingRect(nil, nil, l, b, r, t, e.x, e.y);
    drawingview->Manipulate(e, slidingrect, UpEvent);
    Coord nl, nb, nr, nt;
    slidingrect->GetCurrent(nl, nb, nr, nt);
    delete slidingrect;

    if (nl != l || nb != b) {
      float x0, y0, x1, y1;
      Transformer t;
      drawing->GetPictureTT(t);
      t.InvTransform(float(l), float(b), x0, y0);
      t.InvTransform(float(nl), float(nb), x1, y1);
      Do(new MoveChange(drawing, drawingview, x1 - x0, y1 - y0));
    }
  }
}

// HandleScale lets the user manipulate a scaling rectangle enclosing
// the picked Selection and scales the Selections to the new scale
// when the user releases the button.

void Editor::HandleScale (Event& e) {
  Selection* pick = drawing->PickSelectionIntersecting(e.x, e.y);
  if (pick != nil) {
    drawingview->EraseUngraspedHandles(pick);
    drawing->Grasp(pick);
    drawingview->DrawHandles();

    float l, b, r, t;
    pick->GetBounds(l, b, r, t);
    float cx, cy;
    pick->GetCenter(cx, cy);
    ScalingRect* scalingrect =
    new ScalingRect(nil, nil, round(l), round(b), round(r), round(t),
      round(cx), round(cy));
      drawingview->Manipulate(e, scalingrect, UpEvent);
    float scale = scalingrect->CurrentScaling();
    delete scalingrect;

    if (scale != 0) {
      Do(new ScaleChange(drawing, drawingview, scale, scale));
    }
  }
}
```

```
// HandleModify lets the user modify an already existing Selection
// and replaces the Selection with the modified Selection.  Text
// Selections "modify" themselves using different code below.

void Editor::HandleModify (Event& e) {
 Selection* pick = drawing->PickSelectionShapedBy(e.x, e.y);
 if (pick != nil) {
  drawingview->EraseHandles();
  drawing->Select(pick);
  drawingview->DrawHandles();
  Selection* modifiedpick = nil;
  ClassId cid = pick->GetClassId();
  if (cid == LABEL_OP || cid == LABEL_DF || cid == COMMENT ||
   cid == LAT_DF || cid == MET_OP || cid == LABEL_SL) {
   int len;
   const char* text = ((TextSelection*) pick)->GetOriginal(len);
   TextEdit* textedit = new TextEdit(text, len);
   drawingview->EraseHandles();
   drawingview->Edit(e, textedit, pick);
   text = textedit->GetText(len);
   modifiedpick = new TextSelection(cid, text, len, pick);
   delete textedit;
  }
  else {
   Rubberband* shape = pick->CreateShape(e.x, e.y);
   drawingview->Manipulate(e, shape, UpEvent);
   modifiedpick = pick->GetReshapedCopy();
  }
  if (modifiedpick) {
   Do(new ReplaceChange(drawing, drawingview, pick, modifiedpick));
   drawingview->Draw();
  }
 }
}

// HandleMagnify lets the user manipulate a rubber rectangle and
// expands the given area to fill the view.

void Editor::HandleMagnify (Event& e) {
 RubberRect* rubberrect = NewRubberRectOrSquare(e);
 drawingview->Manipulate(e, rubberrect, UpEvent, false);
 Coord fx, fy;
 rubberrect->GetCurrent(fx, fy, e.x, e.y);
 delete rubberrect;

 drawingview->Magnify(fx, fy, e.x, e.y);
}


// HandleDecompose gives the user of decomposition types and then
// processes the choice

void Editor::HandleDecompose(Event& e) {
 Selection* pick = drawing->PickSelectionIntersecting(e.x, e.y);
 drawingview->EraseHandles();
 if (pick != nil && pick->GetClassId() == LABEL_OP)
  pick = pick->GetOwner()->GetShape();
 if (pick != nil && pick->GetClassId() == OPERATOR) {
  drawing->Select(pick);
  drawingview->DrawHandles();
  char* label = pick->GetOwner()->GetValidLabelString();
  if (label == nil) {
   nolabeldialog->Display();
  } else {
   char result = decomposedialog->Choose();

   // Quit if cancel

   if (result == 'c') return;

   inter->SetCursor(hourglass);
   Save();

   // Construct the child component name
   const char* prototype_name = state->GetDrawingName();
```

```
    char* fixed_label = RemoveBadChars(label);

    char* new_prot_name = new char[strlen(prototype_name)
            + strlen(fixed_label) + 2];
    strcpy(new_prot_name,prototype_name);
                                                    strcat(new_prot_name, ".");
    strcat(new_prot_name,fixed_label);

    char cmd[256];
    char dpopts[128];
    sprintf(dpopts,"-d %s -p %s",dir, new_prot_name);

    int code;
    long status;

    if (result == 'f') {
//    first button pushed, decompose with graphic editor

     if (fork() == 0) {
      if (design_database[0] != NULL)
       code = execlp("a.out", "a.out",
         "-d", dir, "-p", new_prot_name,
         "-c", ddbCmd, "-n", design_database,
         "-geometry", "+450-200", 0);
      else
       code = execlp("a.out", "a.out",
         "-d", dir, "-p", new_prot_name,
         "-c", ddbCmd,
         "-geometry", "+450-200", 0);
      exit(code);
     }
     wait(&status);

     drawing->TagOperator((Vertex *)pick->GetOwner(),"*");
     Save();

    } else {
     // fprintf(stderr, "second or third button chosen.\n");
//    write PSDL IMP file: implementation is Ada


     char* filename = new char[strlen(dir) + strlen(new_prot_name)
          + IMP_EXT_LEN + 1];
     strcpy(filename,dir);
     strcat(filename,new_prot_name);
     strcat(filename,IMP_PSDL_EXT);
     FILE* fptr = fopen(filename, "w");
     fprintf(fptr, "%s%s\n", IMP_ADA_TKN, label);
     fprintf(fptr, "%s", END_TKN);
     fclose(fptr);
     delete [] filename;

     if (result == 's') {
//     start Ada syntax editor

      filename = new char[strlen(dir) + strlen(new_prot_name) + 3];
      strcpy(filename,dir);
      strcat(filename,new_prot_name);
      strcat(filename,".a");
      if (!drawing->Exists(filename)) {
       // file does not exist so create new template
       FILE* fptr = fopen(filename, "w");
       drawing->GenerateAda(fptr,(Vertex *)pick->GetOwner());
       fclose(fptr);
      }
      delete [] filename;
      sprintf(cmd,"ada_syntax_editor.script %s",dpopts);
      int code = system(cmd);
     } else {
      // search for reusable components
      sprintf(cmd,"software_base.script %s",dpopts);
      int code = system(cmd);
     }
    }
    delete new_prot_name;
    inter->SetCursor(defaultCursor);
```

```
    }
  }
}

// HandleText lets the user type some text and creates a new
// TextSelection when the user finishes typing the text.  It must
// clear the selection list because DrawingView will redraw the
// handles obscured by the TextEdit if the list's not empty.

void Editor::HandleText (Event& e) {
 drawingview->EraseHandles();
 drawing->Clear();
 TextEdit* textedit = new TextEdit;
 drawingview->Edit(e, textedit);
 int len;
 const char* text = textedit->GetText(len);
 if (len > 0) {
    TextSelection *ts = new TextSelection(COMMENT, text,
        len, state->GetTextGS());
   drawing->CommentAppend(ts);
   drawing->Select(ts);
   Do(new AddChange(drawing, drawingview));
 }
 delete textedit;
}

// HandleLabel lets the user pick a Selection and adds text in that selection.
// The text will then be centered on the selection.

void Editor::HandleLabel (Event& e) {
 Selection* pick = drawing->PickSelectionIntersecting(e.x, e.y);
 drawingview->EraseHandles();

 if (pick) {
  ClassId cid_pick = pick->GetClassId();
  if (!( (cid_pick == OPERATOR) || (cid_pick == TERMINATOR) ||
       (cid_pick == DATAFLOW_SPLINE) || (cid_pick == SELFLOOP)))
    return;
  drawing->Select(pick);
  drawingview->DrawHandles();

//  can only add label to one selection at a time

  if (drawing->GetNumberOfGraphics() == 1) {
   TextEdit* textedit = new TextEdit;
   drawingview->Edit(e, textedit);
   char *text = textedit->GetString();
   if (strlen(text) > 0) {

//     determine the type of label that we are using

     ClassId cid_text;
     TextSelection *lat = 0;
     if (cid_pick == OPERATOR || cid_pick == TERMINATOR)
       cid_text = LABEL_OP;
     else if (cid_pick == DATAFLOW_SPLINE) {
         cid_text = LABEL_DF;
         lat = ((Edge *) pick->GetOwner())->GetLatency();
       }
     else if (cid_pick == SELFLOOP)
        cid_text  = LABEL_SL;
     TextSelection* ts = new TextSelection(cid_text, text, strlen(text),
              state->GetTextGS());
//     add label to operator list

     pick->GetOwner()->SetLabel(ts);
     drawing->Select(ts);
     Do(new AddChange(drawing, drawingview));

//     center label to selection
     drawing->Select(pick);
     drawing->Extend(ts);
     AlignCenters();

     if (lat) {
       drawing->Select(ts);
```

```
      drawing->Extend(lat);
      AlignCenters();
      AlignTopToBottom();
     }

     drawingview->EraseHandles();
     drawing->Clear();
    }
    delete textedit;
   }
   }
}

// HandleMET letes the user select an operator, add the maximum execution
// time associated with that operator. This MET is place on top of the
// operator.

void Editor::HandleMET (Event& e, char *units ) {
 Selection* pick = drawing->PickSelectionIntersecting(e.x, e.y);
 drawingview->EraseHandles();
 if (pick != nil) {
  if (pick->GetClassId() == LABEL_OP)
     pick = pick->GetOwner()->GetShape();

//  can only add the MET to an operator

  if (pick->GetClassId() == OPERATOR) {
   drawing->Select(pick);
   drawingview->DrawHandles();
   TextEdit* textedit = new TextEdit;
   drawingview->Edit(e, textedit);
   char *orig = textedit->GetString();
   if (strlen(orig) > 0) {
    char *text = new char[strlen(orig) + strlen(units) + 2];
    strcpy(text,orig);
    strcat (text, units);

    TextSelection* ts = new TextSelection(MET_OP, text, strlen(text),
              state->GetTextGS());

//     add MET to operator list

    ((Operator *) pick->GetOwner())->SetMET(ts);
    drawing->Select(ts);
    Do(new AddChange(drawing, drawingview));

//     place MET on top of operator
    drawing->Select(pick);
    drawing->Extend(ts);
    AlignCenters();
    AlignBottomToTop();
    drawingview->EraseHandles();
    drawing->Clear();
   }
   delete textedit;
  }
  }
}

// Adding the TRIGGER_IF condition.
//

void Editor::HandleIf (Event& e ) {
 Selection* pick = drawing->PickSelectionIntersecting(e.x, e.y);
 drawingview->EraseHandles();
 if (pick != nil) {
  if (pick->GetClassId() == LABEL_OP)
     pick = pick->GetOwner()->GetShape();

//  can only add the Trigger_If  to an constraints

  if (pick->GetClassId() == OPERATOR) {
   drawing->Select(pick);
   drawingview->DrawHandles();
   TextEdit* textedit = new TextEdit;
   drawingview->Edit(e, textedit);
```

```
    char *text = textedit->GetString();

    if (strlen(text) > 0) {
     TextSelection* ts = new TextSelection(TRIG_IF_CONS, text, strlen(text),
                state->GetTextGS());

//     add TRIGGER_IF to constraints  list

     drawing->TriggerAppend(ts, (EllipseSelection*) pick);
     drawing->Select(ts);
     Do(new AddChange(drawing, drawingview));


//     place trigger condition on bottom of operator
     drawing->Select(pick);
     drawing->Extend(ts);
     AlignCenters();
     AlignTopToBottom();
     drawingview->EraseHandles();
     drawing->Clear();

    }
    delete textedit;
   }
   }
}


// HandleLatency letes the user select a data flow, add the latency
// associated with that data flow. This latency is placed on top of the
// data flow.

void Editor::HandleLatency (Event& e, char * units ) {
 Selection* pick = drawing->PickSelectionIntersecting(e.x, e.y);
 drawingview->EraseHandles();
 if (pick != nil) {
  if (pick->GetClassId() == LABEL_DF)
   pick = pick->GetOwner()->GetShape();
//  can only add the Latency to stream

  if (pick->GetClassId() == DATAFLOW_SPLINE) {
   Selection *oldpick = pick;
   Selection *l = pick->GetOwner()->GetLabel();
   pick = l ? l : pick;
   drawing->Select(oldpick);
   drawingview->DrawHandles();

   TextEdit* textedit = new TextEdit;
   drawingview->Edit(e, textedit);
   char *orig = textedit->GetString();
   if (strlen(orig) > 0) {
    char *text = new char[strlen(orig) + strlen(units) + 2];
    strcpy(text,orig);
    strcat (text, units);

    TextSelection* ts = new TextSelection(LAT_DF, text, strlen(text),
                state->GetTextGS());

//     add latency to operator list

    ((Edge *) pick->GetOwner())->SetLatency(ts);
    drawing->Select(ts);
    Do(new AddChange(drawing, drawingview));

//     place Latency on top of data flow
      drawing->Select(pick);
      drawing->Extend(ts);
      AlignCenters();
      if (pick != oldpick)
        AlignTopToBottom();
    drawingview->EraseHandles();
    drawing->Clear();
   }
   delete textedit;
  }
  }
```

```
}

// HandleBSpline lets the user draw a series of connected lines and
// creates a BSplineSelection when the user presses the middle button.

void Editor::HandleBSpline (Event& e) {
 Coord* x;
 Coord* y;
 int n;

 drawingview->EraseHandles();
 InputVertices(e, x, y, n);

 if (n != 2 || x[0] != x[1] || y[0] != y[1]) {

// determine which operators the spline is attached to and recompute
// the spline's endpoints to be the intersection of the operator and
// the spline

  Vertex* es0;
  Vertex* esn;
  es0 = drawing->SetEndptsInOperator(x[0],y[0],x[1],y[1]);
  esn = drawing->SetEndptsInOperator(x[n-1],y[n-1],x[n-2],y[n-2]);
  if (es0 != nil || esn != nil) {

//  determine type of spline
    IBrush *oldBrush = state->GetBrush();
    MapIBrush *mb = state->GetMapIBrush();

    if (es0 == esn)
     state->SetBrush(mb->FindOrAppend(0,0xffff,1,0,1));

//  use the arrowhead's pattern to draw the line

    IPattern* temp = state->GetPattern();
    state->SetPattern(state->GetArrowPattern());
    BSplineSelection* ss =
     new BSplineSelection(x, y, n, state->GetGraphicGS());

//  append the spline to the operator list

    drawing->DataFlowSplineAppend(ss, es0, esn);
    drawing->Select(ss);

//  reset the pattern to be the original pattern

    state->SetBrush(oldBrush);
    state->SetPattern(temp);
    Do(new AddChange(drawing, drawingview));
    drawingview->EraseHandles();
   }
  }
 }


// HandleEllipse draws a circle with a radius of 35 pixels at the position
// of the user's mouse when he clicks the left mouse button
// Changed 2/6/91
// By Patrick D. Barnes
// Description: Fixed bug where size of operator is not adjusted for
//  Zoom.


void Editor::HandleEllipse (Event& e) {
 float a00, a01, a10, scale, a20, a21;
 int radius;
 Transformer t;
 drawing->GetPictureTT(t);
 t.GetEntries(a00, a01, a10, scale, a20, a21);
 radius = (int) ((float) OperatorRadius * scale);

 drawingview->EraseHandles();
 state->Constrain(e.x, e.y);
 EllipseSelection* es =
```

```
 new EllipseSelection(e.x, e.y, radius, radius,
                state->GetGraphicGS());
 drawing->Select(es);
 drawing->OperatorAppend(es);
 Do(new AddChange(drawing, drawingview));
}

// HandleRect draws a square with a side of 35 pixels at the position
// of the user's mouse when he clicks the left mouse button
// Changed 11/7/94
// By C.S.Eagle
// Description: Created function

void Editor::HandleRect (Event& e) {
 float a00, a01, a10, scale, a20, a21;
 int offset;
 Transformer t;
 drawing->GetPictureTT(t);
 t.GetEntries(a00, a01, a10, scale, a20, a21);
 offset = (int) ((float) OperatorRadius * scale);

 drawingview->EraseHandles();
 state->Constrain(e.x, e.y);
 RectSelection* rs =
 new RectSelection(e.x-offset, e.y-offset, e.x+offset, e.y+offset,
                state->GetGraphicGS());
 drawing->Select(rs);
 drawing->OperatorAppend(rs);
 Do(new AddChange(drawing, drawingview));
}


// New offers to write an unsaved drawing and creates a new empty
// drawing if the save succeeds or the user refuses the offer.

void Editor::New () {
 boolean successful = OfferToSave();
 if (successful) {
  drawing->ClearPicture();
  Reset(nil);
 }
}

// Revert rereads the drawing from its file.  It asks for confirmation
// before reverting an unsaved drawing.

void Editor::Revert () {
 const char* prototype_name = state->GetDrawingName();
 if (prototype_name != nil) {
  char response = revertdialog->Confirm();
  if (response == 'y') {
   const char* filename = MakeFilename(prototype_name);
    drawing->ReadDFDFiles(dir, prototype_name);
    Reset(prototype_name);
/*    }
   else {
    savecurdialog->
    SetWarning("couldn't revert! (file nonexistent?)");
    Open();
   }*/
  }
 }
}

// Open reads a drawing from a file whose filename is created by the given
// prototype name.  If it fails, it calls the interactive Open to ask the
// user to type another name.

// Modified: 10/19/94 by C.S.Eagle
// to conform to new .graph format

void Editor::Open (const char* prototype_name) {
 drawing->ClearPicture();
 drawing->ReadDFDFiles(dir, prototype_name);
 const char* spec_filename = MakeSpecFilename(prototype_name);
 Reset(prototype_name);
```

```
}

// Open prompts for a prototype name.  It then appends ".graph" to that
// name to make the proper file name and reads a drawing from that file.
// It offers to save an unsaved drawing and it keeps trying to read a
// drawing until it succeeds or the user cancels the command.

void Editor::Open () {
 if (OfferToSave()) {
   drawing->ClearPicture();
   const char* prototype_name = nil;
   char* prototype_array[MAXPROTOTYPES];
   find_prototype_names(prototype_array, dir, "edit");
   opendialog->Clear();
   opendialog->Insert(prototype_array);
   for (;;) {
    prototype_name = opendialog->Select();
    if (prototype_name == nil)
     break;
    else {
     drawing->ReadDFDFiles(dir, prototype_name);
     const char* spec_filename =
        MakeSpecFilename(prototype_name);
     Reset(prototype_name);
     break;
    }
   }
 }
}

// Save writes the drawing to the file it was read from unless there's
// no file or it can't write the drawing to that file, in which case
// it hands the job off to SaveAs.

// Change: 1/29/92
// Author: Pat Barnes
// Description: Disable SaveAs

void Editor::Save () {
 const char* prototype_name = state->GetDrawingName();
 if (prototype_name == nil) {
   if (state->GetModifStatus() == Modified)
     SaveAs();
 }
 else if (state->GetModifStatus() == ReadOnly) {
  saveasdialog->SetErrorTitle("Can't save in read-only file!");
  SaveAs();
 }
 else {
  const char* filename = MakeFilename(prototype_name);
  boolean successful = drawing->WritePicture(filename, state);
  if (successful) {
   state->SetModifStatus(Unmodified);
   state->UpdateViews();
   drawing->WriteDFDFiles(dir,prototype_name);
  }
  else
   saveasdialog->SetErrorTitle("Couldn't save!");
 }
}

// Change: 1/29/92
// Author: Pat Barnes
// Description: Commit calls save to save the files, then calls the
//  Caps 92 UI DatabaseManager function "Execute" to load the
//  subtree into the database. Note: to maintain consistency,
//  the assumption is that any change affects all the children
//  in the operator subtree decomposition.

void Editor::Commit () {
 int code;
 long status;
 char prototype_name[256];
 const char* object_name = state->GetDrawingName();
 if (object_name != nil) {
 Save();
```

```
 if (design_database[0] == NULL) {
  fprintf(stderr, "Error: Cannot commit, database name not set.\n");
  return;
 }
 strcpy(prototype_name, object_name);
 strtok(prototype_name, ".\n\0");
 inter->SetCursor(hourglass);
 if (fork() == 0) {
  code = execlp("design_database",
  ddbCmd,
  design_database,
  "vaa",
  prototype_name,
  object_name, 0);
  exit(code);
 }
 }

 wait(&status);
 inter->SetCursor(defaultCursor);
}


// SaveAs prompts for a prototype name.  It then uses that name to
// determine the file name and writes the drawing to that file.
// It asks for confirmation before overwriting an already existing
// file and it keeps trying to write the drawing until it succeeds or
// the user cancels the command.

void Editor::SaveAs () {
 const char* prototype_name = nil;
 char* prototype_array[MAXPROTOTYPES];
 for (;;) {
  find_prototype_names(prototype_array, dir, "edit");
  saveasdialog->Clear();
  saveasdialog->Insert(prototype_array);
  prototype_name = saveasdialog->Select();
  if (prototype_name == nil)
   break;
  else {
   char filename[128];
   strcpy(filename,prototype_name);
   strcat(filename,DFD_EXT);
   if (drawing->Exists(filename)) {
    overwritedialog->SetWarning("a drawing named ", prototype_name);
    char response = overwritedialog->Confirm();
    if (response != 'y')
     break;
   }
   const char* drawfilename = MakeFilename(prototype_name);
   boolean successful = drawing->WritePicture(drawfilename, state);
   if (successful) {
    state->SetDrawingName(prototype_name);
    state->SetModifStatus(Unmodified);
    state->UpdateViews();
    drawing->WriteDFDFiles(dir,prototype_name);
    break;
   }
   else
    saveasdialog->SetErrorTitle("Couldn't save!");
  }
 }
 saveasdialog->SetErrorTitle("");
}

// Print prompts for a print command and writes the drawing through a
// pipe to that command's standard input.  It keeps trying to print
// the drawing until it succeeds or the user cancels the command.

void Editor::Print () {
 if (state->GetModifStatus() == Modified) {
 savecurdialog->SetWarning("a broken pipe signal won't be caught");
 }
 boolean successful = OfferToSave();
 if (successful) {
 char* cmd = nil;
```

```
for (;;) {
delete cmd;
cmd = printdialog->Edit(nil);
if (cmd == nil) {
break;
}
boolean successful = drawing->PrintPicture(cmd, state);
if (successful) {
break;
} else {
printdialog->SetWarning("couldn't execute ", cmd);
}
}
delete cmd;
}
}

// Skew comments/code ratio to work around cpp bug
;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;
;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;

// Quit offers to save an unsaved drawing and tells Idraw to quit
// running if the save succeeds or the user refuses the offer.

// Modified 2/3/92
// by Patrick D. Barnes
// Description: Change quit to always save

void Editor::Quit (Event& e) {
 //boolean successful = OfferToSave();
 //if (successful) {
 Save();
 if (state->GetModifStatus() == Unmodified) {
 e.target = nil;
 }
}

// Checkpoint writes an unsaved drawing to a temporary filename. The
// program currently calls Checkpoint only when an X error occurs.

void Editor::Checkpoint () {

/*                  extern char* tempnam(char*, char*);                 */

 if (state->GetModifStatus() == Modified) {
/* char* path = tempnam("./", "idraw"); */
 char* path = tmpnam("./idraw");
 boolean successful = drawing->WritePicture(path, state);
 if (successful) {
 fprintf(stderr, "saved drawing as \"%s\"\n", path);
 } else {
 fprintf(stderr, "sorry, couldn't save drawing as \"%s\"\n", path);
 }
 delete path;
 } else {
 fprintf(stderr, "drawing was unmodified, didn't save it\n");
 }
}

// Undo undoes the last change made to the drawing. Undo does nothing
// if all stored changes have been undone.

void Editor::Undo () {
 history->Undo();
}

// Redo redoes the last undone change made to the drawing, i.e., it
// undoes an Undo. Redo does nothing if it follows a Do.

void Editor::Redo () {
 history->Redo();
}

// Cut removes the Selections and writes them to the clipboard file,
// overwriting whatever was there previously.
```

```
void Editor::Cut () {
 Do(new CutChange(drawing, drawingview));
}

// Copy copies the Selections and writes them to the clipboard file,
// overwriting whatever was there previously.

void Editor::Copy () {
 Do(new CopyChange(drawing, drawingview));
}

// Paste reads new Selections from the clipboard file and appends them
// to the drawing.

void Editor::Paste () {
 Do(new PasteChange(drawing, drawingview, state));
}

// Duplicate duplicates the Selections and appends the new Selections
// to the drawing.

void Editor::Duplicate () {
 Do(new DuplicateChange(drawing, drawingview));
}

// Delete deletes all of the Selections.

void Editor::Delete () {
 Do(new DeleteChange(drawing, drawingview));
}

// SelectAll selects all of the Selections in the drawing.

void Editor::SelectAll () {
 drawing->SelectAll();
 drawingview->DrawHandles();
}


// SetBrush sets the Selections' brush and updates the views to
// display the new current brush.

void Editor::SetBrush (IBrush* brush) {
 state->SetBrush(brush);
 state->UpdateViews();
 Do(new SetBrushChange(drawing, drawingview, brush));
}

// Skew comments/code ratio to work around cpp bug
;;;
```

```
 state->SetFont(font);
 state->UpdateViews();
 Do(new SetFontChange(drawing, drawingview, font));
}

// SetPattern sets the Selections' pattern and updates the views to
// display the new current pattern.

void Editor::SetPattern (IPattern* pattern) {
 state->SetPattern(pattern);
 state->UpdateViews();
 Do(new SetPatternChange(drawing, drawingview, pattern));
}

// AlignLeftSides aligns the rest of the Selections's left sides with
// the first Selection's left side.

void Editor::AlignLeftSides () {
 Do(new AlignChange(drawing, drawingview, Left, Left));
}

// AlignRightSides aligns the rest of the Selections' right sides with
// the first Selection's right side.

void Editor::AlignRightSides () {
 Do(new AlignChange(drawing, drawingview, Right, Right));
}

// AlignBottomSides aligns the rest of the Selections' bottom sides
// with the first Selection's bottom side.

void Editor::AlignBottoms () {
 Do(new AlignChange(drawing, drawingview, Bottom, Bottom));
}

// AlignTopSides aligns the rest of the Selections' top sides with the
// first Selection's top side.

void Editor::AlignTops () {
 Do(new AlignChange(drawing, drawingview, Top, Top));
}

// AlignVertCenters aligns the rest of the Selections' vertical
// centers with the first Selection's vertical center.

void Editor::AlignVertCenters () {
 Do(new AlignChange(drawing, drawingview, VertCenter, VertCenter));
}

// AlignHorizCenters aligns the rest of the Selections' horizontal
// centers with the first Selection's horizontal center.

void Editor::AlignHorizCenters () {
 Do(new AlignChange(drawing, drawingview, HorizCenter, HorizCenter));
}

// AlignCenters aligns the rest of the Selections' centers with the
// first Selection's center.

void Editor::AlignCenters () {
 Do(new AlignChange(drawing, drawingview, Center, Center));
}

// AlignLeftToRight aligns each Selection's left side with its
// predecessor's right side.

void Editor::AlignLeftToRight () {
 Do(new AlignChange(drawing, drawingview, Right, Left));
}

// AlignRightToLeft aligns each Selection's right side with its
// predecessor's left side.

void Editor::AlignRightToLeft () {
 Do(new AlignChange(drawing, drawingview, Left, Right));
}
```

```
// AlignBottomToTop aligns each Selection's bottom side with its
// predecessor's top side.

void Editor::AlignBottomToTop () {
  Do(new AlignChange(drawing, drawingview, Top, Bottom));
}

// AlignTopToBottom aligns each Selection's top side with its
// predecessor's bottom side.

void Editor::AlignTopToBottom () {
  Do(new AlignChange(drawing, drawingview, Bottom, Top));
}

// AlignToGrid aligns the Selections' lower left corners with the
// closest grid point.

void Editor::AlignToGrid () {
  Do(new AlignToGridChange(drawing, drawingview));
}

// Reduce reduces the drawing's magnification by a factor of two.

void Editor::Reduce () {
  drawingview->Reduce();
}

// Enlarge enlarges the drawing's magnification by a factor of two.

void Editor::Enlarge () {
  drawingview->Enlarge();
}

// NormalSize resets the drawing's magnification.

void Editor::NormalSize () {
  drawingview->NormalSize();
}

// ReduceToFit reduces the drawing's magnification enough to fit all
// of the drawing in the window.

void Editor::ReduceToFit () {
  drawingview->ReduceToFit();
}

// CenterPage scrolls the drawing so its center coincidences with the
// window's center.

void Editor::CenterPage () {
  drawingview->CenterPage();
}

// RedrawPage redraws the drawing without moving the view.

void Editor::RedrawPage () {
  drawingview->Draw();
}

// GriddingOnOff toggles the grid's constraint on or off.

void Editor::GriddingOnOff () {
  state->SetGridGravity(!state->GetGridGravity());
  state->UpdateViews();
}

// GridVisibleInvisible toggles the grid's visibility on or off.

void Editor::GridVisibleInvisible () {
  state->SetGridVisibility(!state->GetGridVisibility());
  drawingview->Draw();
}

// GridSpacing prompts the user for the new grid spacing in units of
// points. If we didn't use points as units, the same grid spacing
```

```
// would not be portable to different displays.

void Editor::GridSpacing () {
 static char oldspacing[50];
 sprintf(oldspacing, "%lg", state->GetGridSpacing());
 char* spacing = nil;
 for (;;) {
 delete spacing;
 spacing = spacingdialog->Edit(oldspacing);
 if (spacing == nil) {
 break;
 }
 float s = 0.;
 if (sscanf(spacing, "%f", &s) == 1) {
 if (s > 0.) {
 state->SetGridSpacing(s);
 if (state->GetGridVisibility() == true) {
  drawingview->Update();
 }
 }
 break;
 } else {
 spacingdialog->SetWarning("couldn't parse ", spacing);
 }
 }
 delete spacing;
}

// Orientation toggles the page between portrait and landscape
// orientations.

void Editor::Orientation () {
 state->ToggleOrientation();
 drawingview->Update();
}

// ShowVersion displays idraw's version level and author.

void Editor::ShowVersion () {
 versiondialog->Display();
}

// Construct the filename from the given prototype name

const char* Editor::MakeFilename(const char* name) {
 char* filename = new char [strlen(dir) + strlen(name) + GRAPH_EXT_LEN + 1];
 strcpy(filename,dir);
 strcat(filename,name);
 strcat(filename,GRAPH_EXT);
 return filename;
}

const char* Editor::MakeSpecFilename(const char* name) {
  char* filename = new char [strlen(dir) + strlen(name) +
          SPEC_EXT_LEN + 1];
 strcpy(filename,dir);
 strcat(filename,name);
 strcat(filename,SPEC_PSDL_EXT);
 return filename;
}

// Do performs a change to the drawing and updates the drawing's
// modification status if it was unmodified.

void Editor::Do (ChangeNode* changenode) {
 switch (state->GetModifStatus()) {
 case Unmodified:
 history->Do(changenode);
 state->SetModifStatus(Modified);
 state->UpdateViews();
 break;
 default:
 history->Do(changenode);
 break;
 }
}
```

```
// InputVertices lets the user keep drawing a series of connected
// lines until the user presses a button other than the left button.
// It returns the vertices inputted by the user.

void Editor::InputVertices (Event& e, Coord*& xret, Coord*& yret, int& nret) {
 const int INITIALSIZE = 100;
 static int sizebuffers = 0;
 static RubberLine** rubberlines = nil;
 static Coord* x = nil;
 static Coord* y = nil;
 if (INITIALSIZE > sizebuffers) {
  sizebuffers = INITIALSIZE;
  rubberlines = new RubberLine*[sizebuffers];
  x = new Coord[sizebuffers];
  y = new Coord[sizebuffers];
 }

 int n = 0;
 state->Constrain(e.x, e.y);
 rubberlines[0] = nil;
 x[0] = e.x;
 y[0] = e.y;
 ++n;

 while (e.button == LEFTMOUSE) {
 rubberlines[n] = NewRubberLineOrAxis(e);
 e.eventType = UpEvent;
 drawingview->Manipulate(e, rubberlines[n], DownEvent, true, false);
 Coord dummy;
 rubberlines[n]->GetCurrent(dummy, dummy, e.x, e.y);
 x[n] = e.x;
 y[n] = e.y;
 ++n;

 if (n == sizebuffers) {
 RubberLine** oldrubberlines = rubberlines;
 Coord* oldx = x;
 Coord* oldy = y;
 sizebuffers += INITIALSIZE/2;
 rubberlines = new RubberLine*[sizebuffers];
 x = new Coord[sizebuffers];
 y = new Coord[sizebuffers];
 bcopy(oldrubberlines, rubberlines, n * sizeof(RubberLine*));
 bcopy(oldx, x, n * sizeof(Coord));
 bcopy(oldy, y, n * sizeof(Coord));
 delete oldrubberlines;
 delete oldx;
 delete oldy;
 }
 }

 xret = x;
 yret = y;
 nret = n;
 for (int i = 1; i < n; i++) {
 rubberlines[i]->Erase();
 delete rubberlines[i];
 }
}

// NewRubberLineOrAxis creates and returns a new RubberLine or a new
// RubberAxis depending on whether the shift key's being depressed.

RubberLine* Editor::NewRubberLineOrAxis (Event& e) {
 return (!e.shift ?
 new RubberLine(nil, nil, e.x, e.y, e.x, e.y) :
 new RubberAxis(nil, nil, e.x, e.y, e.x, e.y));
}

// NewRubberEllipseOrCircle creates and returns a new RubberEllipse or
// a new RubberCircle depending on the shift key's state.

RubberEllipse* Editor::NewRubberEllipseOrCircle (Event& e) {
 return (!e.shift ?
 new RubberEllipse(nil, nil, e.x, e.y, e.x, e.y) :
```

```
 new RubberCircle(nil, nil, e.x, e.y, e.x, e.y));
}

// Skew comments/code ratio to work around cpp bug
;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;
;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;

// NewRubberRectOrSquare creates and returns a new RubberRect or a new
// RubberSquare depending on whether the shift key's being depressed.

RubberRect* Editor::NewRubberRectOrSquare (Event& e) {
 return (!e.shift ?
 new RubberRect(nil, nil, e.x, e.y, e.x, e.y) :
 new RubberSquare(nil, nil, e.x, e.y, e.x, e.y));
}

// OfferToSave returns true if it saves an unsaved drawing or the user
// refuses the offer or no changes need to be saved.

boolean Editor::OfferToSave () {
 boolean successful = false;
 if (state->GetModifStatus() == Modified) {
 char response = savecurdialog->Confirm();
 if (response == 'y') {
 Save();
 if (state->GetModifStatus() == Unmodified) {
 successful = true;
 }
 } else if (response == 'n') {
 successful = true;
 }
 } else {
 successful = true;
 }
 return successful;
}

// Reset redraws the view, clears the history, and resets state
// information about the drawing's name and its modification status.

void Editor::Reset (const char* prototype_name) {
 history->Clear();
 state->SetDrawingName(prototype_name);
// if (prototype_name != nil && !drawing->Writable(filename)) {
//   state->SetModifStatus(ReadOnly);
// }
// else {
   state->SetModifStatus(Unmodified);
// }
 state->UpdateViews();
 drawingview->Update();
}

// ResetMessage changes the editor's message block

void Editor::ResetMessage(const char* msg) {
 state->SetMessage(msg);
 state->UpdateViews();
}

void Editor::ToolSet(char l) {
  PanelItem *t = tools->LookUp(l);
  t->SetMessage();
  tools->SetCur(t);
}
```

errhandler.h

```
#ifndef errhandler_h
#define errhandler_h

#include <InterViews/reqerr.h>

// Declare imported types.

class Editor;
```

```
// ErrHandler calls upon the Editor to save the current drawing if an
// X request error occurs.

class ErrHandler : public ReqErr {
public:

   ErrHandler();
   ~ErrHandler();

   void SetEditor(Editor*);
   void Error();

protected:

   Editor* editor;                                   // handles drawing and editing operations

};

#endif
```

errhandler.c

```
#include "editor.h"
#include "errhandler.h"
#include <stdio.h>
#include <stdlib.h>

// ErrHandler doesn't have an Editor yet.

ErrHandler::ErrHandler () {
   editor = nil;
}

ErrHandler::~ErrHandler () { }

// SetEditor sets the Editor which ErrHandler calls upon.

void ErrHandler::SetEditor (Editor* e) {
   editor = e;
}

// Error prints the X error, checkpoints the current drawing, and
// terminates the program's execution.

void ErrHandler::Error () {
   fprintf(stderr, "X Error: %s\n", message);
   fprintf(stderr, "      Request code: %d\n", request);
   fprintf(stderr, "      Request function: %d\n", detail);
   fprintf(stderr, "      Request window 0x%x\n", id);
   fprintf(stderr, "      Error Serial #%d\n", msgid);
   if (editor != nil) {
                  editor->Checkpoint();
   }
   const int ERROR = 1;
   exit(ERROR);
}
```

highlighter.h

```
#ifndef highlighter_h
#define highlighter_h

#include <InterViews/scene.h>

// A HighlighterParent creates a highlight painter for its interior
// Highlighters to share.

class HighlighterParent : public MonoScene {
public:

   HighlighterParent();
   ~HighlighterParent();

   boolean SameOutputAs(Painter*);
   Painter* GetHighlightPainter();
```

```
protected:

  Painter* highlight;                                   // stores painter to give interior Highlighters

};

// A Highlighter draws itself and highlights or unhighlights itself on
// command.

class Highlighter : public Interactor {
public:

  Highlighter();
  ~Highlighter();

  void SetHighlighterParent(HighlighterParent*);

  void Highlight(boolean on);

protected:

  void Reconfig();

  HighlighterParent* hparent;// gives us highlight painter if it's nonnil
  boolean highlighted;// stores true if we should be highlighted
  Painter* highlight;                                   // draws us with reversed colors
  Painter* normal;                                      // draws us with normal colors

};

#endif
```

highlighter.c

```
#include "highlighter.h"
#include <InterViews/painter.h>

// HighlighterParent starts with no highlight painter.

HighlighterParent::HighlighterParent () {
  highlight = nil;
}

// Free storage allocated for the highlight painter.

HighlighterParent::~HighlighterParent () {
  Unref(highlight);
}

// SameOutputAs compares the given painter to our output painter so
// our interior Highlighters can decide if they can share our
// highlight painter.

boolean HighlighterParent::SameOutputAs (Painter* out) {
  return out == output;
}

// GetHighlightPainter creates our highlight painter if we don't have
// one yet and returns it so all of our interior Highlighters can
// share it as well as our output painter which they inherit
// automatically. We can't just create highlight in Reconfig because
// our interior Highlighters execute their Reconfig before we do, so
// we create it here (once) when they call us from their Reconfig.

Painter* HighlighterParent::GetHighlightPainter () {
  if (highlight == nil) {
                highlight = new Painter(output);
                highlight->Reference();
                highlight->SetColors(output->GetBgColor(), output->GetFgColor());
  }
  return highlight;
}

// Highlighter starts off unhighlighted with no HighlighterParent yet.
```

```
Highlighter::Highlighter () {
    hparent = nil;
    highlighted = false;
    highlight = nil;
    normal = nil;
}

// Free storage allocated for the highlight painter.

Highlighter::~Highlighter () {
    output = normal;
    Unref(highlight);
}

// SetHighlighterParent gives us a HighlighterParent.

void Highlighter::SetHighlighterParent (HighlighterParent* hp) {
    hparent = hp;
}

// Highlight exchanges our painter and draws us unless we
// don't have a canvas so a panel can highlight us before the panel's
// inserted and a menu can unhighlight us after the menu's removed.

void Highlighter::Highlight (boolean on) {
    highlighted = on;
    output = on ? highlight : normal;
    if (canvas != nil) {
                Redraw(0, 0, xmax, ymax);
    }
}

// Reconfig initializes our highlight painter if necessary by getting
// it from our HighlighterParent if possible or else creating a new
// painter. Then Reconfig switches to the appropriate painter.

void Highlighter::Reconfig () {
    Interactor::Reconfig();
    if (output != highlight && output != normal) {
                Unref(highlight);
                if (hparent != nil && hparent->SameOutputAs(output)) {
                    highlight = hparent->GetHighlightPainter();
                    highlight->Reference();
                } else {                                        // bite the bullet
                    highlight = new Painter(output);
                    highlight->Reference();
                    highlight->SetColors(output->GetBgColor(), output->GetFgColor());
                }
                normal = output;
    }
    output = highlighted ? highlight : normal;
}
```

history.h

```
#ifndef history_h
#define history_h

#include <InterViews/defs.h>

// Declare imported types.

class ChangeNode;
class ChangeList;
class Interactor;

// A History maintains a log of changes made to the drawing to permit
// changes to be undone.

class History {
public:

    History(Interactor*);
    ~History();
```

```
  boolean IsEmpty();
  void Clear();
  void Do(ChangeNode*);
  void Redo();
  void Undo();

protected:

  int maxhistory;                                   // maximum number of changes to store
  ChangeList* changelist;// stores changes to drawing

};

#endif
```

history.c

```
#include "history.h"
#include "istring.h"
#include "listchange.h"
#include <InterViews/interactor.h>
#include <InterViews/Std/stdio.h>

// History sets its maximum number of changes and creates the history.

History::History (Interactor* i) {
  const char* def = i->GetAttribute("history");
  char* definition = strdup(def); // some sscanfs write to their format...
  if (sscanf(definition, "%d", &maxhistory) != 1) {
                maxhistory = 20;                    // default if we can't parse definition
                fprintf(stderr, "can't parse attribute for history, ");
                fprintf(stderr, "value set to %d\n", maxhistory);
  }
  delete definition;
  changelist = new ChangeList;
}

// ~History frees storage allocated for the changes.

History::~History () {
  delete changelist;
}

// IsEmpty returns true if there are no changes stored in the list.

boolean History::IsEmpty () {
  return changelist->Size() == 0;
}

// Clear deletes all of the stored changes.  Calling Clear prevents
// dangling pointers from being referenced if the user tries to undo a
// stored change that was made before a New, Revert, or Open command.

void History::Clear () {
  changelist->DeleteAll();
}

// Do truncates the history at its current point, trims the history to
// its last maxhistory-1 changes, performs the change, appends the
// change, and leaves the current point at the end of the history.

void History::Do (ChangeNode* changenode) {
  while (!changelist->AtEnd()) {
                changelist->DeleteCur();
  }
  changelist->First();
  while (changelist->Size() >= maxhistory) {
                changelist->DeleteCur();
  }
  changenode->Do();
  changelist->Append(changenode);
  changelist->Last();
  changelist->Next();
}

// Redo redoes a stored change in the history and steps over it unless
```

```
// the current point is already at the end of the history.

void History::Redo () {
  if (!changelist->AtEnd()) {
                ChangeNode* changenode = changelist->GetCur();
                changenode->Do();
                changelist->Next();
  }
}

// Undo steps back in the history and undoes a stored change unless
// the current point is already at the beginning of the history.

void History::Undo () {
  changelist->Prev();
  if (!changelist->AtEnd()) {
                ChangeNode* changenode = changelist->GetCur();
                changenode->Undo();
  } else {
                changelist->Next();
  }
}
```

idraw.h

```
#ifndef idraw_h
#define idraw_h

#include <InterViews/scene.h>

// Declare imported types.

class Drawing;
class DrawingView;
class Editor;
class ErrHandler;
class MapKey;
class State;
class Tools;

// An Idraw displays a drawing editor.

class Idraw : public MonoScene {
public:

  Idraw(int, char**);
  ~Idraw();

  void Run();

  void Handle(Event&);
  void Update();

 int wWidth() { return xmax+1; }
 int wHeight() { return ymax+1; }

protected:

  void ParseArgs(int, char**);
  void Init();

  const char* initial_prot;// stores name of initial prototype to use if any
  char* dir;        // stores name of directory of prototypes
  char* ddb;        // stores name of desoign database
  char* ddbCmd;     // stores name of command to load files to ddb

  Drawing* drawing;                    // performs operations on drawing
  DrawingView* drawingview;// displays drawing
  Editor* editor;                      // handles drawing and editing operations
  ErrHandler* errhandler;  // handles an X request error
  MapKey* mapkey;                      // maps characters to Interactors
  State* state;                        // stores current state info about drawing
  Tools* tools;                        // displays drawing tools

};
```

```
#endif
```

Idraw.c

```
#include "commands.h"
#include "drawing.h"
#include "drawingview.h"
#include "editor.h"
#include "errhandler.h"
#include "idraw.h"
#include "istring.h"
#include "mapkey.h"
#include "state.h"
#include "stateviews.h"
#include "tools.h"
#include <InterViews/Graphic/ppaint.h>
#include <InterViews/border.h>
#include <InterViews/box.h>
#include <InterViews/cursor.h>
#include <InterViews/event.h>
#include <InterViews/frame.h>
#include <InterViews/glue.h>
#include <InterViews/panner.h>
#include <InterViews/perspective.h>
#include <InterViews/sensor.h>
#include <InterViews/transformer.h>
#include <InterViews/tray.h>
#include <InterViews/Std/os/fs.h>
#include <sys/param.h>
#include <InterViews/Std/stdio.h>
#include <string.h>
#include <stdlib.h>

// Idraw parses its command line and initializes its members.

Idraw::Idraw (int argc, char** argv) {
   ParseArgs(argc, argv);
   InitPPaint();
   Init();
}

// Free storage allocated for members not in Idraw's scene.

Idraw::~Idraw () {
/*   delete drawing;
   delete dir;
   delete ddb;
   delete ddbCmd;
   delete initial_prot;
   delete editor;
   delete errhandler;
   delete mapkey;
   delete state; */
}

// Run opens the initial file if one was given before starting to run.

void Idraw::Run () {

// pass name of prototype directory to editor class
// Changed 1/31/92
// By Patrick d. Barnes
// added database name and command to pass to editor.

   editor->SetDirectory(dir, ddb, ddbCmd);

   if (initial_prot != nil) {
      SetCursor(hourglass);
      editor->Open(initial_prot);
      SetCursor(defaultCursor);
   }

   Interactor::Run();
}
```

```
// Handle routes keystrokes to their associated interactors.

void Idraw::Handle (Event& e) {
   switch (e.eventType) {
   case KeyEvent:
 if (e.len > 0) {
    Interactor* i = mapkey->LookUp(e.keystring[0]);
    if (i != nil) {
  i->Handle(e);
    }
 }
 break;
   default:
 break;
   }
}

// Update gets the picture's total tranformation matrix whenever it
// changes and stores it in State for creating new graphics.

void Idraw::Update () {
   Transformer t;
   drawing->GetPictureTT(t);
   state->SetGraphicT(t);
}

// ParseArgs stores the name of an initial prototype and name of prototype
// directory to use if any.
// Changed: 1/31/92
// By Patrick D. Barnes
// Rewrote to add flags -c db command, and -n db name.

void Idraw::ParseArgs (int argc, char** argv) {
   dir = nil;
   ddb = "";
   ddbCmd = "design_database";
   initial_prot = nil;
   for (int i=1;i < argc; i++) {
    // fprintf(stderr, "argument:%s; %c\n", argv[i], argv[i][0]);
    if (i >= argc || argv[i][0] != '-') {
     fprintf(stderr, "erroneous argument:%s\n", argv[i]);
     fprintf(stderr, "usage: graphic_editor {[-p prototype_name] [-n database_name]\n");
     fprintf(stderr, "    [-d prototype_dir] [-c database_cmd]}\n");
     const int PARSINGERROR = 1;
     exit(PARSINGERROR);
    }
    switch (argv[i][1]) {
     case 'p':
      initial_prot = argv[++i];
      break;
     case 'd':
      dir = argv[++i];
      break;
     case 'c':
      ddbCmd = argv[++i];
      break;
     case 'n':
      ddb = argv[++i];
      break;
     default:
      fprintf(stderr, "erroneous argument:%s\n", argv[i]);
      fprintf(stderr, "usage: graphic_editor {[-p prototype_name] [-n database_name]\n");
      fprintf(stderr, "    [-d prototype_dir] [-c database_cmd]}\n");
      exit(1);
    }
   }
// set default dir
   if (dir == nil) {

//    use current directory  to search for and save drawing files

     const int bufsize = MAXPATHLEN + 1;
     static char buf[bufsize];
     getcwd(buf,bufsize);
     dir = buf;
   }
```

```
}

// Init creates a sensor to catch keystrokes, creates members and
// initializes links between them, and composes them into a view with
// boxes, borders, glue, and frames.

void Idraw::Init () {
  input = new Sensor;
  input->Catch(KeyEvent);

  drawing    = new Drawing(8.5*inches, 11*inches, 0.05*inch);
  drawingview = new DrawingView(drawing->GetPage());
  editor     = new Editor(this);
  errhandler = new ErrHandler;
  mapkey     = new MapKey;
  state      = new State(this, drawing->GetPage());
  tools      = new Tools(editor, mapkey);

// eagle change for Shing editor
  drawing->SetState(state);
  drawing->SetDrawingView(drawingview);
  drawing->SetIdraw(this);
  drawingview->GetPerspective()->Attach(this);
  drawingview->SetSelectionList(drawing->GetSelectionList());
  drawingview->SetState(state);
  drawingview->SetTools(tools);
  editor->SetDrawing(drawing);
  editor->SetDrawingView(drawingview);
  editor->SetState(state);
  editor->SetTools(tools);
  errhandler->SetEditor(editor);
  errhandler->Install();

  VBox* status = new VBox(
    new HBox(
      new ModifStatusView(state),
      new DrawingNameView(state),
      new GriddingView(state),
      new FontView(state),
      new MagnifView(state, drawingview)
    ),
    new HBorder
  );

  VBox* msgblock = new VBox(
    new HBox(
      new HGlue(10,10),
      new MsgView(state)
    ),
    new HBorder
  );
  HBox* indics = new HBox(
    new BrushView(state),
    new VBorder,
    new PattemView(state)
  );
  HBox* cmds = new HBox(
    new Commands(editor, mapkey, state),

//    pad drawing area with 200 extra pixels

    new HGlue(200)
  );

  VBox* panel = new VBox(
    tools,
    new VGlue,
    new HBorder,
    new Panner(drawingview)
  );
  panel->Propagate(false);

  HBorder* hborder = new HBorder;
  VBorder* vborder = new VBorder;

  Tray* t = new Tray;
```

```
    t->HBox(t, status, t);
    t->HBox(t, msgblock, t);
    t->HBox(t, indics, vborder, cmds, t);
    t->HBox(t, hborder, t);
    t->HBox(t, panel, vborder, drawingview, t);

    t->VBox(t, status, msgblock, indics, hborder, panel, t);
    t->VBox(t, status, msgblock, vborder, t);
    t->VBox(t, status, msgblock, cmds, hborder, drawingview, t);

    Insert(new Frame(t, 1));
}
```

iellipses.h

```
#ifndef iellipses_h
#define iellipses_h

#include <InterViews/Graphic/ellipses.h>

// An IFillEllipse knows when NOT to draw itself.

class IFillEllipse : public FillEllipse {
public:

    IFillEllipse(Coord, Coord, int, int, Graphic* = nil);

protected:

    boolean contains(PointObj&, Graphic*);
    boolean intersects(BoxObj&, Graphic*);
    void draw(Canvas*, Graphic*);

};

// An IFillCircle knows when NOT to draw itself.

class IFillCircle : public FillCircle {
public:

    IFillCircle(Coord, Coord, int, Graphic* = nil);

protected:

    boolean contains(PointObj&, Graphic*);
    boolean intersects(BoxObj&, Graphic*);
    void draw(Canvas*, Graphic*);

};

#endif
```

iellipses.c

```
#include "iellipses.h"
#include "ipaint.h"

// IFillEllipse passes its arguments to FillEllipse.

IFillEllipse::IFillEllipse (Coord x0, Coord y0, int rx, int ry, Graphic* gs)
: (x0, y0, rx, ry, gs) {
}

// contains returns true if the IFillEllipse contains the given point
// unless the pattern is the "none" pattern.

boolean IFillEllipse::contains (PointObj& po, Graphic* gs) {
    boolean contains = false;
    IPattern* pattern = (IPattern*) gs->GetPattern();
    if (!pattern->None()) {
contains = FillEllipse::contains(po, gs);
    }
    return contains;
}
```

```
// intersects returns true if the IFillEllipse intersects the given
// box unless the pattern is the "none" pattern.

boolean IFillEllipse::intersects (BoxObj& userb, Graphic* gs) {
    boolean intersects = false;
    IPattern* pattern = (IPattern*) gs->GetPattern();
    if (!pattern->None()) {
intersects = FillEllipse::intersects(userb, gs);
    }
    return intersects;
}

// draw draws the IFillEllipse unless the pattern is the "none" pattern.

void IFillEllipse::draw (Canvas* c, Graphic* gs) {
    IPattern* pattern = (IPattern*) gs->GetPattern();
    if (!pattern->None()) {
FillEllipse::draw(c, gs);
    }
}

// IFillCircle passes its arguments to FillCircle.

IFillCircle::IFillCircle (Coord x0, Coord y0, int r, Graphic* gs)
: (x0, y0, r, gs) {
}

// contains returns true if the IFillCircle contains the given point
// unless the pattern is the "none" pattern.

boolean IFillCircle::contains (PointObj& po, Graphic* gs) {
    boolean contains = false;
    IPattern* pattern = (IPattern*) gs->GetPattern();
    if (!pattern->None()) {
contains = FillCircle::contains(po, gs);
    }
    return contains;
}

// intersects returns true if the IFillCircle intersects the given box
// unless the pattern is the "none" pattern.

boolean IFillCircle::intersects (BoxObj& userb, Graphic* gs) {
    boolean intersects = false;
    IPattern* pattern = (IPattern*) gs->GetPattern();
    if (!pattern->None()) {
intersects = FillCircle::intersects(userb, gs);
    }
    return intersects;
}

// draw draws the IFillCircle unless the pattern is the "none"
// pattern.

void IFillCircle::draw (Canvas* c, Graphic* gs) {
    IPattern* pattern = (IPattern*) gs->GetPattern();
    if (!pattern->None()) {
FillCircle::draw(c, gs);
    }
}
```

ilines.h

```
#ifndef ilines_h
#define ilines_h

#include <InterViews/Graphic/polygons.h>

// An IFillMultiLine knows when NOT to draw itself.

class IFillMultiLine : public FillPolygon {
public:

    IFillMultiLine(Coord*, Coord*, int, Graphic* = nil);
```

```
protected:

   boolean contains(PointObj&, Graphic*);
   boolean intersects(BoxObj&, Graphic*);
   void draw(Canvas*, Graphic*);

};

#endif
```

ilines.c

```
#include "ilines.h"
#include "ipaint.h"

// IFillMultiLine passes its arguments to FillPolygon.

IFillMultiLine::IFillMultiLine (Coord* x, Coord* y, int n, Graphic* gs)
: (x, y, n, gs) {
}

// contains returns true if the IFillMultiLine contains the given point
// unless the brush is an arrow or the pattern is the "none" pattern.

boolean IFillMultiLine::contains (PointObj& po, Graphic* gs) {
   boolean contains = false;
   IBrush* brush = (IBrush*) gs->GetBrush();
   IPattern* pattern = (IPattern*) gs->GetPattern();
   if (!brush->LeftArrow() && !brush->RightArrow() && !pattern->None()) {
contains = FillPolygon::contains(po, gs);
   }
   return contains;
}

// intersects returns true if the IFillMultiLine intersects the given
// box unless the brush is an arrow or the pattern is the "none"
// pattern.

boolean IFillMultiLine::intersects (BoxObj& userb, Graphic* gs) {
   boolean intersects = false;
   IBrush* brush = (IBrush*) gs->GetBrush();
   IPattern* pattern = (IPattern*) gs->GetPattern();
   if (!brush->LeftArrow() && !brush->RightArrow() && !pattern->None()) {
intersects = FillPolygon::intersects(userb, gs);
   }
   return intersects;
}

// draw draws the IFillMultiLine unless the brush is an arrow or the
// pattern is the "none" pattern (a IFillPolygon wouldn't have cared
// about the brush being an arrow).

void IFillMultiLine::draw (Canvas* c, Graphic* gs) {
   IBrush* brush = (IBrush*) gs->GetBrush();
   IPattern* pattern = (IPattern*) gs->GetPattern();
   if (!brush->LeftArrow() && !brush->RightArrow() && !pattern->None()) {
FillPolygon::draw(c, gs);
   }
}
```

ipaint.h

```
#ifndef ipaint_h
#define ipaint_h

#include <InterViews/Graphic/ppaint.h>

// An IBrush knows how to test for its noneness and get its line
// pattern, width, arrows, and dash pattern.

class IBrush : public PBrush {
public:

   IBrush();
   IBrush(int, int, boolean, boolean);

   boolean None();
   int GetLinePattern();
```

```
    int Width();
    boolean LeftArrow();
    boolean RightArrow();
    const int* GetDashPattern();
    int GetDashPatternSize();
    int GetDashOffset();
    operator Brush*();

protected:

    void CalcDashPat(int);

    boolean leftarrow;// stores true if line starts from an arrowhead
    boolean rightarrow;// stores true if line ends in an arrowhead
    int dashpat[patternWidth];// stores dash pattern
    int dashpatsize;// stores number of defined elements in dashpat
    int dashoffset;// stores dash pattern's offset

};

// Define inline access functions to get members' values.

inline boolean IBrush::None () {
    return (value == nil);
}

inline boolean IBrush::LeftArrow () {
    return leftarrow;
}

inline boolean IBrush::RightArrow () {
    return rightarrow;
}

inline int IBrush::GetLinePattern () {
    return p;
}

inline const int* IBrush::GetDashPattern () {
    return dashpat;
}

inline int IBrush::GetDashPatternSize () {
    return dashpatsize;
}

inline int IBrush::GetDashOffset () {
    return dashoffset;
}

inline IBrush::operator Brush* () {
    return value;
}

// An IColor knows how to get its name.

class IColor : public PColor {
public:

    IColor(const char*);
    IColor(int, int, int, const char*);
    IColor(Color*, const char*);
    ~IColor();

    const char* GetName();
    operator Color*();

protected:

    char* name;// stores name passed into constructor

};

// Define inline access functions to get members' values.

inline const char* IColor::GetName () {
```

```
    return name;
}

inline IColor::operator Color* () {
    return value;
}

// An IFont knows how to get its name, print font, and print size.

class IFont : public PFont {
public:

    IFont(const char*, const char*, const char*);
    ~IFont();

    const char* GetName();
    const char* GetPrintFont();
    const char* GetPrintSize();
    const char* GetPrintFontAndSize();
    int GetLineHt();
    operator Font*();

protected:

    const char* FilterName(const char*);

    char* printfont;// stores name of font used by printer
    char* printsize;// stores scale of font used by printer
    char* printfontandsize;// stores name and size separated by a blank
    int lineHt;// stores printsize converted to int

};

// Define inline access functions to get members' values.

inline const char* IFont::GetName () {
    return name ? name : "stdfont";
}

inline const char* IFont::GetPrintFont () {
    return printfont;
}

inline const char* IFont::GetPrintSize () {
    return printsize;
}

inline const char* IFont::GetPrintFontAndSize () {
    return printfontandsize;
}

inline int IFont::GetLineHt () {
    return lineHt;
}

inline IFont::operator Font* () {
    return value;
}

// An IPattern knows how to test for its noneness or fullness and get
// its dither, data, and gray level.

class IPattern : public PPattern {
public:

    IPattern();
    IPattern(int, float);
    IPattern(int pattern[patternHeight], int);

    boolean None();
    float GetGrayLevel();
    const int* GetData();
    int GetSize();
    operator Pattern*();

protected:
```

```
    float graylevel;// stores gray level for grayscale patterns
    int size;// stores pat's orig size (4x4, 8x8, or 16x16)

};

// Define inline access functions to get members' values.

inline boolean IPattern::None () {
    return (value == nil);
}

inline float IPattern::GetGrayLevel () {
    return graylevel;
}

inline const int* IPattern::GetData () {
    return data;
}

inline int IPattern::GetSize () {
    return size;
}

inline IPattern::operator Pattern* () {
    return value;
}

#endif
```

ipaint.c

```
 #include "ipaint.h"
#include "istring.h"
#include <stdlib.h>

// IBrush creates the brush.  Calling IBrush with no arguments creates
// the "none" brush.

IBrush::IBrush () : () {
    leftarrow = false;
    rightarrow = false;
    CalcDashPat(0xffff);
}

IBrush::IBrush (int p, int w, boolean l, boolean r) : (p, w) {
    leftarrow = l;
    rightarrow = r;
    CalcDashPat(p);
}

// Width overrides PBrush::Width when PBrush's value is the predefined
// brush single because single cannot be trusted to return 1.  It
// might return 0 because the X implementation might support only a
// fast, device-dependent brush, not the standard single-width brush.

int IBrush::Width () {
    int width = PBrush::Width();
    if (value == single) {
width = 1;
    }
    return width;
}

// CalcDashPat calculates and stores the Postscript dash pattern
// corresponding to the brush's line pattern.

void IBrush::CalcDashPat (int linepat) {
    linepat &= 0xffff;// mask should always match patternWidth

    if (linepat == 0x0000) {    // clear brush
dashpatsize = 0;
dashoffset = -1;
    } else if (linepat == 0xffff) { // solid brush
dashpatsize = 0;
dashoffset = 0;
```

```
    } else if (linepat == 0x5555) { // dotted brush, store 1 element not 16
dashpat[0] = 1;
dashpatsize = 1;
dashoffset = 1;
    } else if (linepat == 0xaaaa) { // dotted brush, store 1 element not 16
dashpat[0] = 1;
dashpatsize = 1;
dashoffset = 0;
    } else {
int i = 0;
while (!((linepat << i) & 0x8000)) {
    ++i;
}
dashoffset = patternWidth - i + 1;

int j = 0;
boolean currentrun = true;
int length = 0;
for (int k = 0; k < patternWidth; k++) {
    if ((((linepat << i) & 0x8000) != 0) == currentrun) {
++length;
    } else {
dashpat[j++] = length;
currentrun = !currentrun;
length = 1;
    }
    i = (i == patternWidth) ? 0 : i + 1;
}
if (length > 0) {
    dashpat[j] = length;
}
dashpatsize = j + 1;
    }

    const int Postscriptdashlimit = 11;
    if (dashpatsize > Postscriptdashlimit) {
fprintf(stderr, "Brush dash pattern 0x%x exceeds maximum ", linepat);
fprintf(stderr, "length of Postscript dash pattern with ");
fprintf(stderr, "%d elements, truncated to ", dashpatsize);
fprintf(stderr, "%d elements\n", Postscriptdashlimit);
dashpatsize = Postscriptdashlimit;
    }
}

// IColor creates the named color and stores its name.

IColor::IColor (const char* n) : (n) {
    name = strdup(n);
}

// IColor creates the color using the given intensities and stores its
// "name".

IColor::IColor (int r, int g, int b, const char* n) : (r, g, b) {
    name = strdup(n);
}

// IColor stores the given color and its "name".

IColor::IColor (Color* color, const char* n) {
    value = color;
    value->Reference();
    name = strdup(n);
}

// Free storage allocated for the name.

IColor::~IColor () {
    delete name;
}

// IFont creates the named font and stores the print font and size.

IFont::IFont (const char* name, const char* pf, const char* ps)
: (FilterName(name)) {
    printfont = strdup(pf);
```

```
    printsize = strdup(ps);
    printfontandsize = new char[strlen(pf) + 1 + strlen(ps) + 1];
    strcpy(printfontandsize, pf);
    strcat(printfontandsize, " ");
    strcat(printfontandsize, ps);
    lineHt = atoi(ps);
}

// Free storage allocated for the print font and size.

IFont::~IFont () {
    delete printfont;
    delete printsize;
    delete printfontandsize;
}

// FilterName filters the name to ensure "stdfont" does not pass
// through to PFont without being converted to nil.

const char* IFont::FilterName (const char* name) {
    if (strcmp(name, "stdfont") == 0) {
name = nil;
    }
    return name;
}

// IPattern creates the pattern.  Calling IPattern with no arguments
// creates the "none" pattern, calling IPattern with an integer
// creates a pattern from a 4x4 bitmap, calling IPattern with an array
// and the actual size creates a pattern from a 16x16 bitmap that may
// have originally been 8x8, and calling IPattern with an integer and
// a float creates a grayscale pattern from a 4x4 bitmap.

IPattern::IPattern () : () {
    graylevel = -1;
    size = 0;
}

IPattern::IPattern (int dither, float g) : (dither) {
    graylevel = g;
    size = 0;
}

IPattern::IPattern (int data[patternHeight], int s) : (data) {
    graylevel = -1;
    size = s;
}
```

ipolygons.h

```
#ifndef ipolygons_h
#define ipolygons_h

#include <InterViews/Graphic/polygons.h>

// An IFillRect knows when NOT to draw itself.

class IFillRect : public FillRect {
public:

    IFillRect(Coord, Coord, Coord, Coord, Graphic* = nil);

protected:

    boolean contains(PointObj&, Graphic*);
    boolean intersects(BoxObj&, Graphic*);
    void draw(Canvas*, Graphic*);

};

// An IFillPolygon knows when NOT to draw itself.

class IFillPolygon : public FillPolygon {
public:
```

```
    IFillPolygon(Coord*, Coord*, int, Graphic* = nil);

  protected:

    boolean contains(PointObj&, Graphic*);
    boolean intersects(BoxObj&, Graphic*);
    void draw(Canvas*, Graphic*);

  };

  #endif
```

ipolygons.c

```
  #include "ipaint.h"
  #include "ipolygons.h"

  // IFillRect passes its arguments to FillRect.

  IFillRect::IFillRect (Coord x0, Coord y0, Coord x1, Coord y1, Graphic* gs)
  : (x0, y0, x1, y1, gs) {
  }

  // contains returns true if the IFillRect contains the given point
  // unless the pattern is the "none" pattern.

  boolean IFillRect::contains (PointObj& po, Graphic* gs) {
    boolean contains = false;
    IPattern* pattern = (IPattern*) gs->GetPattern();
    if (!pattern->None()) {
  contains = FillRect::contains(po, gs);
    }
    return contains;
  }

  // intersects returns true if the IFillRect intersects the given box
  // unless the pattern is the "none" pattern.

  boolean IFillRect::intersects (BoxObj& userb, Graphic* gs) {
    boolean intersects = false;
    IPattern* pattern = (IPattern*) gs->GetPattern();
    if (!pattern->None()) {
  intersects = FillRect::intersects(userb, gs);
    }
    return intersects;
  }

  // draw draws the IFillRect unless the pattern is the "none" pattern.

  void IFillRect::draw (Canvas* c, Graphic* gs) {
    IPattern* pattern = (IPattern*) gs->GetPattern();
    if (!pattern->None()) {
  FillRect::draw(c, gs);
    }
  }

  // IFillPolygon passes its arguments to FillPolygon.

  IFillPolygon::IFillPolygon (Coord* x, Coord* y, int n, Graphic* gs)
  : (x, y, n, gs) {
  }

  // contains returns true if the IFillPolygon contains the given point
  // unless the pattern is the "none" pattern.

  boolean IFillPolygon::contains (PointObj& po, Graphic* gs) {
    boolean contains = false;
    IPattern* pattern = (IPattern*) gs->GetPattern();
    if (!pattern->None()) {
  contains = FillPolygon::contains(po, gs);
    }
    return contains;
  }

  // intersects returns true if the IFillPolygon intersects the given
  // box unless the pattern is the "none" pattern.
```

```
boolean IFillPolygon::intersects (BoxObj& userb, Graphic* gs) {
   boolean intersects = false;
   IPattern* pattern = (IPattern*) gs->GetPattern();
   if (!pattern->None()) {
intersects = FillPolygon::intersects(userb, gs);
   }
   return intersects;
}

// draw draws the IFillPolygon unless the pattern is the "none"
// pattern.

void IFillPolygon::draw (Canvas* c, Graphic* gs) {
   IPattern* pattern = (IPattern*) gs->GetPattern();
   if (!pattern->None()) {
FillPolygon::draw(c, gs);
   }
}
```

isplines.h

```
#ifndef isplines_h
#define isplines_h

#include <InterViews/Graphic/splines.h>

// An IFillBSpline knows when NOT to draw itself.

class IFillBSpline : public FillBSpline {
public:

   IFillBSpline(Coord*, Coord*, int, Graphic* = nil);

protected:

   boolean contains(PointObj&, Graphic*);
   boolean intersects(BoxObj&, Graphic*);
   void draw(Canvas*, Graphic*);

};

// An IFillClosedBSpline knows when NOT to draw itself.

class IFillClosedBSpline : public FillBSpline {
public:

   IFillClosedBSpline(Coord*, Coord*, int, Graphic* = nil);

protected:

   boolean contains(PointObj&, Graphic*);
   boolean intersects(BoxObj&, Graphic*);
   void draw(Canvas*, Graphic*);

};

#endif
```

isplines.c

```
#define _POSIX_SOURCE
#include "ipaint.h"
#include "isplines.h"
#include <InterViews/Std/math.h>
#include <InterViews/Graphic/util.h>

// IFillBSpline repeats the first and last points three times to
// ensure that the spline will pass through the first and last points.

IFillBSpline::IFillBSpline (Coord* ax, Coord* ay, int n, Graphic* gs)
: (ax, ay, n, gs) {
   delete x;
```

```
    delete y;
    count = n + 4;
    x = new Coord[count];
    y = new Coord[count];
    x[0] = ax[0];
    x[1] = ax[0];
    x[count - 1] = ax[n - 1];
    x[count - 2] = ax[n - 1];
    y[0] = ay[0];
    y[1] = ay[0];
    y[count - 1] = ay[n - 1];
    y[count - 2] = ay[n - 1];
    CopyArray(ax, ay, n, &x[2], &y[2]);
}

// contains returns true if the IFillBSpline contains the given point
// unless the brush is an arrow or the pattern is the "none" pattern.

boolean IFillBSpline::contains (PointObj& po, Graphic* gs) {
    boolean contains = false;
    IBrush* brush = (IBrush*) gs->GetBrush();
    IPattern* pattern = (IPattern*) gs->GetPattern();
    if (!brush->LeftArrow() && !brush->RightArrow() && !pattern->None()) {
contains = FillBSpline::contains(po, gs);
    }
    return contains;
}

// intersects returns true if the IFillBSpline intersects the given
// box unless the brush is an arrow or the pattern is the "none"
// pattern.

boolean IFillBSpline::intersects (BoxObj& userb, Graphic* gs) {
    boolean intersects = false;
    IBrush* brush = (IBrush*) gs->GetBrush();
    IPattern* pattern = (IPattern*) gs->GetPattern();
    if (!brush->LeftArrow() && !brush->RightArrow() && !pattern->None()) {
intersects = FillBSpline::intersects(userb, gs);
    }
    return intersects;
}

// draw draws the IFillBSpline unless the brush is an arrow or the
// pattern is the "none" pattern.

void IFillBSpline::draw (Canvas* c, Graphic* gs) {
    IBrush* brush = (IBrush*) gs->GetBrush();
    IPattern* pattern = (IPattern*) gs->GetPattern();
    if (!brush->LeftArrow() && !brush->RightArrow() && !pattern->None()) {
FillBSpline::draw(c, gs);
    }
}

// IFillClosedBSpline passes its arguments to FillBSpline.

IFillClosedBSpline::IFillClosedBSpline (Coord* x, Coord* y, int n, Graphic* gs)
: (x, y, n, gs) {
}

// contains returns true if the IFillClosedBSpline contains the given
// point unless the pattern is the "none" pattern.

boolean IFillClosedBSpline::contains (PointObj& po, Graphic* gs) {
    boolean contains = false;
    IPattern* pattern = (IPattern*) gs->GetPattern();
    if (!pattern->None()) {
contains = FillBSpline::contains(po, gs);
    }
    return contains;
}

// intersects returns true if the IFillClosedBSpline intersects the
// given box unless the pattern is the "none" pattern.

boolean IFillClosedBSpline::intersects (BoxObj& userb, Graphic* gs) {
    boolean intersects = false;
```

```
    IPattern* pattern = (IPattern*) gs->GetPattern();
    if (!pattern->None()) {
intersects = FillBSpline::intersects(userb, gs);
    }
    return intersects;
}

// draw draws the IFillClosedBSpline unless the pattern is the "none"
// pattern.

void IFillClosedBSpline::draw (Canvas* c, Graphic* gs) {
    IPattern* pattern = (IPattern*) gs->GetPattern();
    if (!pattern->None()) {
FillBSpline::draw(c, gs);
    }
}
```

istring.h

```
#ifndef istring_h
#define istring_h

#include <string.h>

// removes improper characters from a char string

char* RemoveBadChars(char*);
int LengthWithoutChars(char*, char*);

// create tempoary file name by retrieving environment variable
// TEMP and concantenating the filename onto it.

char* MakeTmpFileName(char*);

// strdup allocates and returns a duplicate of the given string.

inline char* strdup (const char* s) {
    char* dup = new char[strlen(s) + 1];
    strcpy(dup, s);
    return dup;
}

// strndup allocates and returns a duplicate of the first len
// characters of the given string.

inline char* strndup (const char* s, int len) {
    char* dup = new char[len + 1];
    strncpy(dup, s, len);
    dup[len] = '\0';
    return dup;
}

#endif
```

istring.c

```
#include "istring.h"
#include <InterViews/defs.h>

extern "C"
{
    extern char *getenv(char *);
}

char* RemoveBadChars(char* string) {
int new_len = LengthWithoutChars(string, "\n");
int old_len = strlen(string);
char* modified_string = new char[old_len + 1];
if (new_len < old_len) {
int m_ctr = 0;
for (int i = 0; i < old_len; ++i) {
if (string[i] != '\n' && string[i] != ' ') {
modified_string[m_ctr++] = string[i];
        }
    }
modified_string[m_ctr] = '\0';
```

```
    }
else {
strcpy(modified_string,string);
    }
return modified_string;
}


int LengthWithoutChars(char* string, char* unwanted_chars) {
 int size = strlen(unwanted_chars);
 int string_size = strlen(string);
 int no_of_chars = 0;
 for (int i = 0; i < size; ++ i) {
for (int j = 0; j < string_size; ++j) {
if (string[j] == unwanted_chars[i]) {
++no_of_chars;
          }
        }
    }
 return string_size - no_of_chars;
}

// make temporary filename by concatenating the environment variable
// TEMP with the given filename

char* MakeTmpFileName(char* filename) {
char* tmp_dir = (char*) getenv("TEMP");
char* result;

if (tmp_dir != nil) {
int tmp_len = strlen(tmp_dir);
result = new char[tmp_len + strlen(filename) + 2];
strcpy(result, tmp_dir);
if(tmp_dir[tmp_len - 1] != '/') {
strcat(result, "/");
      }
strcat(result, filename);
    }
else {
result = filename;
    }
return result;
}
```

keystrokes.h

```
static const char SELECTCHAR        = 's';
static const char MOVECHAR          = 'm';
static const char STATESCHAR        = 'X';
static const char EXCEPTCHAR        = '\005';  //^E';
static const char KEYWORDCHAR       = 'K';
static const char LABELCHAR         = 'L';
static const char LATENCYCHAR       = 'j';

static const char IFCHAR            = 'F';
static const char BYALLCHAR         = 'A';
static const char BYSOMECHAR        = 'Y';

static const char RESET_TIMERCHAR     = ' ';
static const char START_TIMERCHAR     = ' ';
static const char STOP_TIMERCHAR      = ' ';
static const char INFORMALCHAR        = ' ';
static const char FORMALCHAR          = ' ';

static const char HOURCHAR          = 'H';
static const char MINUTECHAR        = 'N';
static const char SECONDCHAR        = 'S';
static const char MILSECCHAR        = 'M';
static const char MICSECCHAR        = 'C';

static const char LHOURCHAR         = '1';
static const char LMINUTECHAR       = '2';
static const char LSECONDCHAR       = '3';
static const char LMILSECCHAR       = '4';
```

```
static const char LMICSECCHAR        = '5';

static const char INTEGERCHAR        = 'i';
static const char REALCHAR           = 'r';
static const char BOOLEANCHAR        = 'b';
static const char USERDEFINEDCHAR    = 'u';
//static const char STRETCHCHAR= ';';
static const char MODIFYCHAR= 'q';
static const char DECOMPOSECHAR      = 'D';
static const char COMMENTCHAR= 'T';
static const char BSPLINECHAR= 'h';
static const char ELLIPSECHAR= 'o';
static const char RECTCHAR= 'r';

// You can execute a PullDownMenuCommand by typing one of these
// characters.

static const char NEWCHAR= '\016'; // ^N
static const char REVERTCHAR= '\022'; // ^R
static const char OPENCHAR= '\017'; // ^O
static const char SAVECHAR= '\023'; // ^S
static const char SAVEASCHAR= '\001'; // ^A
static const char PRINTCHAR= '\020'; // ^P
static const char QUITCHAR= '\021'; // ^Q

static const char UNDOCHAR= 'U';
static const char REDOCHAR= 'R';
static const char DELETECHAR= '\004'; // ^D
static const char SELECTALLCHAR= 'a';

static const char REDUCECHAR= 'f';
static const char ENLARGECHAR= 'e';
static const char NORMALSIZECHAR= 'n';
static const char REDUCETOFITCHAR= '=';
static const char CENTERPAGECHAR= '/';
static const char REDRAWPAGECHAR= '\014'; // ^L
static const char GRIDDINGONOFFCHAR= ',';
static const char GRIDVISIBLEINVISIBLECHAR = '?';
static const char GRIDSPACINGCHAR= 'Z';
static const char ORIENTATIONCHAR= '+';
static const char SHOWVERSIONCHAR= '$';
```

list.h

```
#ifndef list_h
#define list_h

#include <InterViews/defs.h>

// A BaseNode contains links to its previous and next BaseNodes.
// BaseLists can link BaseNodes to each other to form lists of
// BaseNodes.

class BaseNode {

   friend class BaseList;

public:

   BaseNode();
   virtual ~BaseNode();

   virtual boolean SameValueAs(void*);

protected:

private:

   BaseNode* prev;// points to previous BaseNode in list
   BaseNode* next;// points to next BaseNode in list

};

// A BaseList maintains and iterates through a list of BaseNodes.

class BaseList {
public:
```

```
    BaseList();
    ~BaseList();

    int Size();
    boolean AtEnd();
    BaseNode* First();
    BaseNode* Last();
    BaseNode* Prev();
    BaseNode* Next();
    BaseNode* GetCur();
    BaseNode* Index(int);
    boolean Find(void*);

    void Append(BaseNode*);
    void Prepend(BaseNode*);
    void InsertAfterCur(BaseNode*);
    void InsertBeforeCur(BaseNode*);

    void RemoveCur();
    void DeleteCur();
    void DeleteAll();

  private:

    BaseNode* head;// points to dummy head of circular list
    BaseNode* cur;// points to current node of circular list
    int size;// stores number of non-dummy nodes in list

  };

  // Size returns the number of non-dummy nodes in a list.

  inline int BaseList::Size () {
    return size;
  }

  // AtEnd returns true if the current node is the dummy node.

  inline boolean BaseList::AtEnd () {
    return cur == head;
  }

  // First returns the first node in a list (or the head if it's empty)
  // and sets the current node to that node.

  inline BaseNode* BaseList::First () {
    cur = head->next;
    return cur;
  }

  // Last returns the last node in a list (or the head if it's empty)
  // and sets the current node to that node.

  inline BaseNode* BaseList::Last () {
    cur = head->prev;
    return cur;
  }

  // Prev returns the node before the current node and sets the current
  // node to that node.

  inline BaseNode* BaseList::Prev () {
    cur = cur->prev;
    return cur;
  }

  // Next returns the node after the current node and sets the current
  // node to that node.

  inline BaseNode* BaseList::Next () {
    cur = cur->next;
    return cur;
  }

  // GetCur returns the current node in a list.
```

```
inline BaseNode* BaseList::GetCur () {
  return cur;
}

#endif
```

list.c

```
#include "list.h"

// BaseNode zeroes its pointers.

BaseNode::BaseNode () {
  prev = nil;
  next = nil;
}

BaseNode::~BaseNode () {
  // no storage allocated by base class
}

// SameValueAs returns true if this class contains a data member of
// any pointer type that has the same value as the given pointer.

boolean BaseNode::SameValueAs (void*) {
  // base class contains no data members
  return false;
}

// BaseList starts with only a header node.

BaseList::BaseList () {
  cur = head = new BaseNode;
  head->prev = head->next = head;
  size = 0;
}

// ~BaseList destroys the list.

BaseList::~BaseList () {
  DeleteAll();
  delete head;
}

// Index returns the node that would be the indexed element if the
// list was a C array (0 = first, 1 = next, ..., size-1 = last) or nil
// if the index is out of bounds.  Index sets the current node to the
// indexed node if it's in the list; otherwise, the current node
// remains the same.

BaseNode* BaseList::Index (int index) {
  BaseNode* elem = nil;
  if (index >= 0 && index < size) {
elem = head->next;
for (int i = 0; i < index; i++) {
  elem = elem->next;
}
cur = elem;
  }
  return elem;
}

// Find returns true if the list contains a node with a data member
// whose value is the same as the given pointer or false if there is
// no such node.  Find sets the current node to that node only if it
// finds such a node; otherwise, the current node remains the same.

boolean BaseList::Find (void* value) {
  for (BaseNode* elem = head->next; elem != head; elem = elem->next) {
if (elem->SameValueAs(value)) {
  cur = elem;
  return true;
}
  }
  return false;
```

```
}

// Append inserts a node after the last node of the list.  The current
// node remains the same.

void BaseList::Append (BaseNode* appendee) {
BaseNode* last = head->prev;
   appendee->prev = last;
   appendee->next = head;
   head->prev = appendee;
   last->next = appendee;
   ++size;
}

// Prepend inserts a node before the first node of the list.  The
// current node remains the same.

void BaseList::Prepend (BaseNode* prependee) {
   BaseNode* first = head->next;
   prependee->prev = head;
   prependee->next = first;
   first->prev = prependee;
   head->next  = prependee;
   ++size;
}

// InsertAfterCur inserts a node after the current node of the list.
// The current node remains the same.

void BaseList::InsertAfterCur (BaseNode* prependee) {
   BaseNode* first = cur->next;
   prependee->prev = cur;
   prependee->next = first;
   first->prev = prependee;
   cur->next   = prependee;
   ++size;
}

// InsertBeforeCur inserts a node before the current node of the list.
// The current node remains the same.

void BaseList::InsertBeforeCur (BaseNode* appendee) {
   BaseNode* last = cur->prev;
   appendee->prev = last;
   appendee->next = cur;
   cur->prev  = appendee;
   last->next = appendee;
   ++size;
}

// RemoveCur removes the current node of the list (if it's a node and
// not the head) and sets the current node to the following node.

void BaseList::RemoveCur () {
   if (cur != head) {
BaseNode* before = cur->prev;
BaseNode* after  = cur->next;
after->prev  = before;
before->next = after;
cur = after;
--size;
   }
}

// DeleteCur deletes the current node of the list (if it's a node and
// not the head) and sets the current node to the following node.

void BaseList::DeleteCur () {
   if (cur != head) {
BaseNode* before = cur->prev;
BaseNode* after  = cur->next;
after->prev  = before;
before->next = after;
delete cur;
cur = after;
--size;
```

```
  }
}

// DeleteAll deletes all the nodes of the list, making the list empty
// again, and sets the current node to the list's head.

void BaseList::DeleteAll () {
  BaseNode* after = nil;
  for (BaseNode* doomed = head->next; doomed != head; doomed = after) {
after = doomed->next;
delete doomed;
  }
  cur = head;
  head->prev = head->next = head;
  size = 0;
}
```

listboolean.h

```
#ifndef listboolean_h
#define listboolean_h

#include "list.h"

// A booleanNode contains a boolean value.

class booleanNode : public BaseNode {
public:

  booleanNode (boolean v) { value = v; }
  boolean GetBoolean () { return value; }

protected:

  boolean value;// contains a boolean value

};

// A booleanList manages a list of booleanNodes.

class booleanList : public BaseList {
public:

  booleanNode* First();
  booleanNode* Last();
  booleanNode* Prev();
  booleanNode* Next();
  booleanNode* GetCur();
  booleanNode* Index(int);

};

// Cast these functions to return booleanNodes instead of BaseNodes.

inline booleanNode* booleanList::First () {
  return (booleanNode*) BaseList::First();
}

inline booleanNode* booleanList::Last () {
  return (booleanNode*) BaseList::Last();
}

inline booleanNode* booleanList::Prev () {
  return (booleanNode*) BaseList::Prev();
}

inline booleanNode* booleanList::Next () {
  return (booleanNode*) BaseList::Next();
}

inline booleanNode* booleanList::GetCur () {
  return (booleanNode*) BaseList::GetCur();
}

inline booleanNode* booleanList::Index (int index) {
  return (booleanNode*) BaseList::Index(index);
```

```
}

#endif
```

listcenter.h

```
#ifndef listcenter_h
#define listcenter_h

#include "list.h"

// A CenterNode contains two float values.

class CenterNode : public BaseNode {
public:

   CenterNode (float x, float y) { cx = x; cy = y; }
   float GetCx () { return cx; }
   float GetCy () { return cy; }

protected:

   float cx, cy;// stores a position

};

// A CenterList manages a list of CenterNodes.

class CenterList : public BaseList {
public:

   CenterNode* First();
   CenterNode* Last();
   CenterNode* Prev();
   CenterNode* Next();
   CenterNode* GetCur();
   CenterNode* Index(int);

};

// Cast these functions to return CenterNodes instead of BaseNodes.

inline CenterNode* CenterList::First () {
   return (CenterNode*) BaseList::First();
}

inline CenterNode* CenterList::Last () {
   return (CenterNode*) BaseList::Last();
}

inline CenterNode* CenterList::Prev () {
   return (CenterNode*) BaseList::Prev();
}

inline CenterNode* CenterList::Next () {
   return (CenterNode*) BaseList::Next();
}

inline CenterNode* CenterList::GetCur () {
   return (CenterNode*) BaseList::GetCur();
}

inline CenterNode* CenterList::Index (int index) {
   return (CenterNode*) BaseList::Index(index);
}

#endif
```

listchange.h

```
#ifndef listchange_h
#define listchange_h

#include "list.h"
```

```
// Declare imported types.

class CenterList;
class Drawing;
class DrawingView;
class GroupList;
class IBrush;
class IBrushList;
class IColor;
class IColorList;
class IFont;
class IFontList;
class IPattern;
class IPatternList;
class Selection;
class SelectionList;
class State;

// ChangeNode stores a change to the Drawing in reversible form and
// can carry out or remove the change.

class ChangeNode : public BaseNode {
public:

   ChangeNode();
   ChangeNode(Drawing*, DrawingView*, boolean = false);
   ~ChangeNode();

   virtual void Do();
   virtual void Undo();

protected:

   Drawing* drawing;// performs operations on drawing
   DrawingView* drawingview;// displays drawing
   SelectionList* oldsl;// lists the old Selections

};

// MoveChange translates the Selections.

class MoveChange : public ChangeNode {
public:

   MoveChange(Drawing*, DrawingView*, float, float);

   void Do();
   void Undo();

protected:

   float dx, dy;// carries out translation
   float undodx, undody;// removes translation

};

// ScaleChange scales the Selections.

class ScaleChange : public ChangeNode {
public:

   ScaleChange(Drawing*, DrawingView*, float, float);

   void Do();
   void Undo();

protected:

   float sx, sy;// carries out scaling
   float undosx, undosy;// removes scaling

};

// StretchChange stretches the Selections.

class StretchChange : public ChangeNode {
```

```
public:

    StretchChange(Drawing*, DrawingView*, float, Alignment);

    void Do();
    void Undo();

protected:

    Alignment OppositeSide(Alignment);

    float stretch;// carries out stretching
    float undostretch;// removes stretching
    Alignment side;// indicates fixed side for do
    Alignment undoside;// indicates fixed side for undo

};

// RotateChange rotates the Selections.

class RotateChange : public ChangeNode {
public:

    RotateChange(Drawing*, DrawingView*, float);

    void Do();
    void Undo();

protected:

    float angle;// carries out rotation
    float undoangle;// removes rotation

};

// ReplaceChange replaces a Selection with another Selection.

class ReplaceChange : public ChangeNode {
public:

    ReplaceChange(Drawing*, DrawingView*, Selection*, Selection*);
    ~ReplaceChange();

    void Do();
    void Undo();

protected:

    Selection* replacee;// stores to-be-replaced Selection's address
    Selection* replacer;// stores replacing Selection's address

};

// SetBrushChange sets the Selections' brush.

class SetBrushChange : public ChangeNode {
public:

    SetBrushChange(Drawing*, DrawingView*, IBrush*);
    ~SetBrushChange();

    void Do();
    void Undo();

protected:

    IBrush* brush;// brush value to set
    IBrushList* undobrushlist;// brush values to restore

};

// SetFgColorChange sets the Selections' foreground color.

class SetFgColorChange : public ChangeNode {
public:
```

```
    SetFgColorChange(Drawing*, DrawingView*, IColor*);
    ~SetFgColorChange();

    void Do();
    void Undo();

protected:

    IColor* fgcolor;// color value to set
    IColorList* undofglist;// color values to restore

};

// SetBgColorChange sets the Selections' background color.

class SetBgColorChange : public ChangeNode {
public:

    SetBgColorChange(Drawing*, DrawingView*, IColor*);
    ~SetBgColorChange();

    void Do();
    void Undo();

protected:

    IColor* bgcolor;// color value to set
    IColorList* undobglist;// color values to restore

};

// SetFontChange sets the Selections' font.

class SetFontChange : public ChangeNode {
public:

    SetFontChange(Drawing*, DrawingView*, IFont*);
    ~SetFontChange();

    void Do();
    void Undo();

protected:

    IFont* font;// font value to set
    IFontList* undofontlist;// font values to restore

};

// SetPatternChange sets the Selections' pattern.

class SetPatternChange : public ChangeNode {
public:

    SetPatternChange(Drawing*, DrawingView*, IPattern*);
    ~SetPatternChange();

    void Do();
    void Undo();

protected:

    IPattern* pattern;   // pattern value to set
    IPatternList* undopatternlist; // pattern values to restore

};

// AddChange adds the Selections to the Drawing.

class AddChange : public ChangeNode {
public:

    AddChange(Drawing*, DrawingView*);
    ~AddChange();

    void Do();
```

```
    void Undo();

protected:

    boolean done;// remembers if change was done

};

// DeleteChange deletes the Selections from the Drawing.

class DeleteChange : public ChangeNode {
public:

    DeleteChange(Drawing*, DrawingView*);
    ~DeleteChange();

    void Do();
    void Undo();

protected:

    SelectionList* prevlist;// lists the selections' predecessors
    boolean done;// remembers if change was done

};

// CutChange removes the Selections from the Drawing and copies them
// to the Clipboard, deleting the Clipboard's previous contents.

class CutChange : public DeleteChange {
public:

    CutChange(Drawing*, DrawingView*);

    void Do();

};

// CopyChange copies the Selections to the Clipboard, deleting the
// Clipboard's previous contents.

class CopyChange : public ChangeNode {
public:

    CopyChange(Drawing*, DrawingView*);

    void Do();

};

// PasteChange copies the Selections in the Clipboard and appends the
// new Selections to the Drawing.

class PasteChange : public AddChange {
public:

    PasteChange(Drawing*, DrawingView*, State*);

};

// DuplicateChange copies the Selections and appends the new
// Selections to the Drawing.

class DuplicateChange : public AddChange {
public:

    DuplicateChange(Drawing*, DrawingView*);

};

// GroupChange groups the Selections into a single PictSelection.

class GroupChange : public ChangeNode {
public:

    GroupChange(Drawing*, DrawingView*);
```

```
    ~GroupChange();

    void Do();
    void Undo();

protected:

    SelectionList* prevlist;// lists the selections' predecessors
    GroupList* grouplist;// lists the selections and their new parent
    boolean done;// remembers whether change was done

};

// UngroupChange ungroups each PictSelection into its children.

class UngroupChange : public ChangeNode {
public:

    UngroupChange(Drawing*, DrawingView*);
    ~UngroupChange();

    void Do();
    void Undo();

protected:

    GroupList* undogrouplist;// lists the selections and their children
    boolean done;// remembers whether change was done

};

// BringToFrontChange brings the Selections to the front of the
// Drawing.

class BringToFrontChange : public ChangeNode {
public:

    BringToFrontChange(Drawing*, DrawingView*);
    ~BringToFrontChange();

    void Do();
    void Undo();

protected:

    SelectionList* prevlist;// lists the selections' predecessors

};

// SendToBackChange sends the Selections to the back of the Drawing.

class SendToBackChange : public ChangeNode {
public:

    SendToBackChange(Drawing*, DrawingView*);
    ~SendToBackChange();

    void Do();
    void Undo();

protected:

    SelectionList* prevlist;// lists the selections' predecessors

};

// AlignChange aligns the Selections.

class AlignChange : public ChangeNode {
public:

    AlignChange(Drawing*, DrawingView*, Alignment, Alignment);
    ~AlignChange();

    void Do();
    void Undo();
```

```
protected:

   Alignment falign;// part of fixed Selection to align up with
   Alignment malign;// part of moving Selection to align up with
   CenterList* centerlist;// stores Selections' original positions

};

// AlignToGridChange aligns the Selections to the grid.

class AlignToGridChange : public ChangeNode {
public:

   AlignToGridChange(Drawing*, DrawingView*);
   ~AlignToGridChange();

   void Do();
   void Undo();

protected:

   CenterList* centerlist;// stores Selections' original positions

};

// A ChangeList manages a list of ChangeNodes.

class ChangeList : public BaseList {
public:

   ChangeNode* First();
   ChangeNode* Last();
   ChangeNode* Prev();
   ChangeNode* Next();
   ChangeNode* GetCur();
   ChangeNode* Index(int);

};

inline ChangeNode* ChangeList::First () {
   return (ChangeNode*) BaseList::First();
}

inline ChangeNode* ChangeList::Last () {
   return (ChangeNode*) BaseList::Last();
}

inline ChangeNode* ChangeList::Prev () {
   return (ChangeNode*) BaseList::Prev();
}

inline ChangeNode* ChangeList::Next () {
   return (ChangeNode*) BaseList::Next();
}

inline ChangeNode* ChangeList::GetCur () {
   return (ChangeNode*) BaseList::GetCur();
}

inline ChangeNode* ChangeList::Index (int index) {
   return (ChangeNode*) BaseList::Index(index);
}

#endif
```

listchange.c

```
#include "drawing.h"
#include "drawingview.h"
#include "listcenter.h"
#include "listchange.h"
#include "listgroup.h"
#include "listibrush.h"
#include "listicolor.h"
#include "listifont.h"
```

```
#include "listipattern.h"
#include "listselectn.h"
#include "selection.h"
#include "slpict.h"

// ChangeNode creates a dummy ChangeNode object for the header of the
// ChangeList.

ChangeNode::ChangeNode () {
  drawing = nil;
  drawingview = nil;
  oldsl = nil;
}

// ChangeNode stores the Drawing and DrawingView. It copies the
// SelectionList, optionally sorting the SelectionList first.

ChangeNode::ChangeNode (Drawing* d, DrawingView* dv, boolean sort) {
  drawing = d;
  drawingview = dv;
  if (sort) {
drawing->Sort();
  }
  oldsl = drawing->GetSelections();
}

// Free storage allocated for the list of old Selections.

ChangeNode::~ChangeNode () {
  delete oldsl;
}

// Do carries out the change to the Drawing.

void ChangeNode::Do () {
  // nop
}

// Undo removes the change to the Drawing.

void ChangeNode::Undo () {
  // nop
}

// MoveChange stores the Selections' translation in reversible form.

MoveChange::MoveChange (Drawing* d, DrawingView* dv, float xdisp, float ydisp)
: ChangeNode(d, dv) {
  dx = xdisp;
  dy = ydisp;
  undodx = -xdisp;
  undody = -ydisp;
}

// Do moves the Selections.

void MoveChange::Do () {
  drawingview->EraseExcessHandles(oldsl);
  drawing->Select(oldsl);
  drawingview->Damaged();
  drawing->Move(dx, dy, drawingview);
 }

// Undo moves the Selections back to their original places.

void MoveChange::Undo () {
  drawingview->EraseExcessHandles(oldsl);
  drawing->Select(oldsl);
  drawingview->Damaged();
  drawing->Move(undodx, undody, drawingview);
  drawingview->Damaged();
  drawingview->Repair();
}

// ScaleChange stores the Selections' scaling in reversible form.
```

```
ScaleChange::ScaleChange (Drawing* d, DrawingView* dv, float xsc, float ysc)
: ChangeNode(d, dv) {
   sx = xsc;
   sy = ysc;
   undosx = 1.0 / xsc;
   undosy = 1.0 / ysc;
}

// Do scales the Selections.

void ScaleChange::Do () {
   drawingview->EraseExcessHandles(oldsl);
   drawing->Select(oldsl);
   drawingview->Damaged();
   drawing->Scale(sx, sy);
   drawingview->Damaged();
   drawingview->Repair();
}

// Undo scales the Selections back to their former sizes.

void ScaleChange::Undo () {
   drawingview->EraseExcessHandles(oldsl);
   drawing->Select(oldsl);
   drawingview->Damaged();
   drawing->Scale(undosx, undosy);
   drawingview->Damaged();
   drawingview->Repair();
}

// StretchChange stores the Selections' stretching in reversible form.

StretchChange::StretchChange (Drawing* d, DrawingView* dv, float str,
Alignment sd) : ChangeNode(d, dv) {
   stretch = str;
   undostretch = 1/str;
   side = sd;
   undoside = (str < 0) ? OppositeSide(sd) : sd;
}

// Do stretches the Selections.

void StretchChange::Do () {
   drawingview->EraseExcessHandles(oldsl);
   drawing->Select(oldsl);
   drawingview->Damaged();
   drawing->Stretch(stretch, side);
   drawingview->Damaged();
   drawingview->Repair();
}

// Undo stretches the Selections back to their former sizes.

void StretchChange::Undo () {
   drawingview->EraseExcessHandles(oldsl);
   drawing->Select(oldsl);
   drawingview->Damaged();
   drawing->Stretch(undostretch, undoside);
   drawingview->Damaged();
   drawingview->Repair();
}

// OppositeSide returns the side opposite the given side.

Alignment StretchChange::OppositeSide (Alignment original) {
   Alignment opposite;
   switch (original) {
   case Left:
opposite = Right;
break;
   case Right:
opposite = Left;
break;
   case Bottom:
opposite = Top;
break;
```

```
    case Top:
opposite = Bottom;
break;
    }
    return opposite;
}

// RotateChange stores the Selections' rotation in reversible form.

RotateChange::RotateChange (Drawing* d, DrawingView* dv, float a) :
      ChangeNode(d, dv) {
    angle = a;
    undoangle = -a;
}

// Do rotates the Selections.

void RotateChange::Do () {
    drawingview->EraseExcessHandles(oldsl);
    drawing->Select(oldsl);
    drawingview->Damaged();
    drawing->Rotate(angle);
    drawingview->Damaged();
    drawingview->Repair();
}

// Undo rotates the Selections back to their original places.

void RotateChange::Undo () {
    drawingview->EraseExcessHandles(oldsl);
    drawing->Select(oldsl);
    drawingview->Damaged();
    drawing->Rotate(undoangle);
    drawingview->Damaged();
    drawingview->Repair();
}

// Skew comments/code ratio to work around cpp bug
;;;
```

```
// SetBrushChange stores the Selections' original brushes and the new
// brush to set.

SetBrushChange::SetBrushChange (Drawing* d, DrawingView* dv, IBrush* br)
: ChangeNode(d, dv) {
   brush = br;
   undobrushlist = drawing->GetBrush();
}

// Free storage allocated for the list of original brushes.

SetBrushChange::~SetBrushChange () {
   delete undobrushlist;
}

// Do sets the Selections' new brush.

void SetBrushChange::Do () {
   drawingview->EraseExcessHandles(oldsl);
   drawing->Select(oldsl);
   drawingview->Damaged();
   drawing->SetBrush(brush);
   drawingview->Damaged();
   drawingview->Repair();
}

// Undo restores the Selections' original brushes.

void SetBrushChange::Undo () {
   drawingview->EraseExcessHandles(oldsl);
   drawing->Select(oldsl);
   drawingview->Damaged();
   drawing->SetBrush(undobrushlist);
   drawingview->Damaged();
   drawingview->Repair();
}

// SetFgColorChange stores the Selections' original foreground colors
// and the new foreground color to set.

SetFgColorChange::SetFgColorChange (Drawing* d, DrawingView* dv, IColor* fg)
: ChangeNode(d, dv) {
   fgcolor = fg;
   undofglist = drawing->GetFgColor();
}

// Free storage allocated for the list of original colors.

SetFgColorChange::~SetFgColorChange () {
   delete undofglist;
}

// Do sets the Selections' new foreground color.

void SetFgColorChange::Do () {
   drawingview->EraseExcessHandles(oldsl);
   drawing->Select(oldsl);
   drawingview->Damaged();
   drawing->SetFgColor(fgcolor);
   drawingview->Damaged();
   drawingview->Repair();
}

// Undo restores the Selections' original foreground colors.

void SetFgColorChange::Undo () {
   drawingview->EraseExcessHandles(oldsl);
   drawing->Select(oldsl);
   drawingview->Damaged();
   drawing->SetFgColor(undofglist);
   drawingview->Damaged();
   drawingview->Repair();
}

// Skew comments/code ratio to work around cpp bug
;;;;;;;;;;;;;;;;;;;;;;;;;;;;
```

```
// SetBgColorChange stores the Selections' original background colors
// and the new background color to set.

SetBgColorChange::SetBgColorChange (Drawing* d, DrawingView* dv, IColor* bg)
: ChangeNode(d, dv) {
   bgcolor = bg;
   undobglist = drawing->GetBgColor();
}.

// Free storage allocated for the list of original colors.

SetBgColorChange::~SetBgColorChange () {
   delete undobglist;
}

// Do sets the Selections' new background color.

void SetBgColorChange::Do () {
   drawingview->EraseExcessHandles(oldsl);
   drawing->Select(oldsl);
   drawingview->Damaged();
   drawing->SetBgColor(bgcolor);
   drawingview->Damaged();
   drawingview->Repair();
}

// Undo restores the Selections' original background colors.

void SetBgColorChange::Undo () {
   drawingview->EraseExcessHandles(oldsl);
   drawing->Select(oldsl);
   drawingview->Damaged();
   drawing->SetBgColor(undobglist);
   drawingview->Damaged();
   drawingview->Repair();
}

// SetFontChange stores the Selections' original fonts and
// the new font to set.

SetFontChange::SetFontChange (Drawing* d, DrawingView* dv, IFont* f)
: ChangeNode(d, dv) {
   font = f;
   undofontlist = drawing->GetFont();
}

// Free storage allocated for the list of original fonts.

SetFontChange::~SetFontChange () {
   delete undofontlist;
}

// Do sets the Selections' new font.

void SetFontChange::Do () {
   drawingview->EraseExcessHandles(oldsl);
   drawing->Select(oldsl);
   drawingview->Damaged();
   drawing->SetFont(font);
   drawingview->Damaged();
   drawingview->Repair();
}

// Undo restores the Selections' original fonts.

void SetFontChange::Undo () {
   drawingview->EraseExcessHandles(oldsl);
   drawing->Select(oldsl);
   drawingview->Damaged();
   drawing->SetFont(undofontlist);
   drawingview->Damaged();
   drawingview->Repair();
}

// SetPatternChange stores the Selections' original patterns and the
```

```
// new pattern to set.

SetPattemChange::SetPattemChange (Drawing* d, DrawingView* dv, IPattem* pat)
: ChangeNode(d, dv) {
   pattem = pat;
   undopatternlist = drawing->GetPattern();
}

// Free storage allocated for the list of original patterns.

SetPattemChange::~SetPattemChange () {
   delete undopatternlist;
}

// Do sets the Selections' new pattern.

void SetPattemChange::Do () {
   drawingview->EraseExcessHandles(oldsl);
   drawing->Select(oldsl);
   drawing->SetPattem(pattem);
   drawingview->Damaged();
   drawingview->Repair();
}

// Undo restores the Selections' original pattems.

void SetPattemChange::Undo () {
   drawingview->EraseExcessHandles(oldsl);
   drawing->Select(oldsl);
   drawing->SetPattem(undopatternlist);
   drawingview->Damaged();
   drawingview->Repair();
}

// AddChange knows it hasn't done its change yet.

AddChange::AddChange (Drawing* d, DrawingView* dv) : ChangeNode(d, dv) {
   done = false;
}

// Free storage allocated for the Selections if AddChange
// never added them to the Drawing.

AddChange::~AddChange () {
   if (!done) {
for (oldsl->First(); !oldsl->AtEnd(); oldsl->Next()) {
   Selection* s = oldsl->GetCur()->GetSelection();
   delete s;
}
   }
}

// Do appends the Selections to the Drawing.

void AddChange::Do () {
   drawingview->EraseHandles();
   drawing->Select(oldsl);
   drawing->Append();
   drawingview->Added();
   drawingview->Repair();
   done = true;
}

// Undo removes the Selections from the Drawing.

void AddChange::Undo () {
   drawingview->EraseExcessHandles(oldsl);
   drawing->Select(oldsl);
   drawingview->Damaged();
   drawing->Remove();
   drawing->Clear();
   drawingview->Repair();
   done = false;
}

// DeleteChange stores the Selections' predecessors.
```

```
DeleteChange::DeleteChange (Drawing* d, DrawingView* dv) :
  ChangeNode(d, dv, true) {
  prevlist = drawing->GetPrevs();
  done = false;
}

// Free storage allocated for the Selections if DeleteChange
// removed them from the Drawing.

DeleteChange::~DeleteChange () {
  delete prevlist;
  if (done) {
for (oldsl->First(); !oldsl->AtEnd(); oldsl->Next()) {
  Selection* s = oldsl->GetCur()->GetSelection();
  delete s;
}
  }
}

// Do removes the Selections from the Drawing.

void DeleteChange::Do () {
  drawingview->EraseExcessHandles(oldsl);
  drawing->Select(oldsl);
  drawingview->Damaged();
  drawing->Remove();
  drawing->Clear();
  drawingview->Repair();
drawingview->Draw();
  done = true;
}

// Undo puts the Selections back where they came from in the Drawing.

void DeleteChange::Undo () {
  drawingview->EraseExcessHandles(oldsl);
  drawing->Select(oldsl);
  drawing->InsertAfterPrev(prevlist);
  drawingview->Damaged();
  drawingview->Repair();
  done = false;
}

// CutChange passes its arguments to its DeleteChange constructor.

CutChange::CutChange (Drawing* d, DrawingView* dv) : DeleteChange(d, dv) {
}

// Do removes the Selections from the Drawing and writes them to the
// clipboard file, overwriting the clipboard file's previous contents.

void CutChange::Do () {
  drawingview->EraseExcessHandles(oldsl);
  drawing->Select(oldsl);
  drawing->WriteClipboard();
  DeleteChange::Do();
}

// CopyChange must sort the Selections.

CopyChange::CopyChange (Drawing* d, DrawingView* dv) :
ChangeNode(d, dv, true) {
}

// Do writes the Selections to the clipboard file, overwriting
// whatever was there previously.

void CopyChange::Do () {
  drawingview->EraseExcessHandles(oldsl);
  drawing->Select(oldsl);
  drawing->WriteClipboard();
}

// PasteChange reads the clipboard file and stores the clippings for
// pasting into the Drawing later.
```

```
PasteChange::PasteChange (Drawing* d, DrawingView* dv, State* state)
: AddChange(d, dv) {
   delete oldsl;
   oldsl = drawing->ReadClipboard(state);
}

// DuplicateChange stores duplicates of the picked Selections for
// pasting into the Drawing later.

DuplicateChange::DuplicateChange (Drawing* d, DrawingView* dv) :
AddChange(d, dv) {
   drawing->Sort();
   delete oldsl;
   oldsl = drawing->GetDuplicates();
}

// GroupChange stores the Selections' new parent and their
// predecessors.

GroupChange::GroupChange (Drawing* d, DrawingView* dv) :
ChangeNode(d, dv, true) {
   grouplist = drawing->GetParent();
   prevlist = drawing->GetPrevs();
   done = false;
}

// Delete frees storage allocated for the Selections' new parent if
// they never end up grouped under it and the lists themselves.

GroupChange::~GroupChange () {
   if (!done) {
for (grouplist->First(); !grouplist->AtEnd(); grouplist->Next()) {
   PictSelection* s = grouplist->GetCur()->GetParent();
   delete s;
}
   }
   delete grouplist;
   delete prevlist;
}

// Do groups the Selections under their parent.

void GroupChange::Do () {
   drawingview->EraseHandles();
   drawing->Group(grouplist);
   drawingview->Damaged();
   drawingview->Repair();
   done = true;
}

// Undo ungroups the Selections and puts them back where they came
// from in the Drawing.

void GroupChange::Undo () {
   drawingview->EraseHandles();
   drawing->Ungroup(grouplist);
   drawing->Remove();
   drawing->InsertAfterPrev(prevlist);
   drawingview->Damaged();
   drawingview->Repair();
   done = false;
}

// UngroupChange stores the Selections' children.

UngroupChange::UngroupChange (Drawing* d, DrawingView* dv) :
ChangeNode(d, dv, true) {
   undogrouplist = drawing->GetChildren();
   done = false;
}

// Delete frees storage allocated for the Selections if they were
// ungrouped and for the list itself.

UngroupChange::~UngroupChange () {
```

```
  if (done) {
for (undogrouplist->First(); !undogrouplist->AtEnd();
   undogrouplist->Next())
{
  boolean haschildren = undogrouplist->GetCur()->GetHasChildren();
  if (haschildren) {
PictSelection* s = undogrouplist->GetCur()->GetParent();
delete s;
  }
}
  }
  delete undogrouplist;
}

// Do replaces all Selections which contain children with their
// children.

void UngroupChange::Do () {
  drawingview->EraseHandles();
  drawing->Ungroup(undogrouplist);
  drawingview->RedrawHandles();
  done = true;
}

// Undo regroups each set of children under their former parents.

void UngroupChange::Undo () {
  drawingview->EraseHandles();
  drawing->Group(undogrouplist);
  drawingview->RedrawHandles();
  done = false;
}

// Skew comments/code ratio to work around cpp bug
;;;
```

```
: ChangeNode(d, dv, true) {
   prevlist = drawing->GetPrevs();
}

// Delete frees storage allocated for the Selections' predecessors.

SendToBackChange::~SendToBackChange () {
   delete prevlist;
}

// Do sends the Selections to the back by removing them from and
// prepending them to the Drawing.

void SendToBackChange::Do () {
   drawingview->EraseExcessHandles(oldsl);
   drawing->Select(oldsl);
   drawing->Remove();
   drawing->Prepend();
   drawingview->Damaged();
   drawingview->Repair();
}

// Undo puts the Selections back where they came from in the Drawing.

void SendToBackChange::Undo () {
   drawingview->EraseExcessHandles(oldsl);
   drawing->Select(oldsl);
   drawing->Remove();
   drawing->InsertAfterPrev(prevlist);
   drawingview->Damaged();
   drawingview->Repair();
}

// AlignChange stores the Selections' original positions and their
// desired alignments.

AlignChange::AlignChange (Drawing* d, DrawingView* dv, Alignment fix,
Alignment move) : ChangeNode(d, dv) {
   falign = fix;
   malign = move;
   centerlist = drawing->GetCenter();
}

// Delete frees storage allocated for the list of original positions.

AlignChange::~AlignChange () {
   delete centerlist;
}

// Do aligns the Selections.

void AlignChange::Do () {
   drawingview->EraseExcessHandles(oldsl);
   drawing->Select(oldsl);
   drawingview->Damaged();
   drawing->Align(falign, malign);
   drawingview->Damaged();
   drawingview->Repair();
}

// Undo moves the Selections to their original positions.

void AlignChange::Undo () {
   drawingview->EraseExcessHandles(oldsl);
   drawing->Select(oldsl);
   drawingview->Damaged();
   drawing->SetCenter(centerlist);
   drawingview->Damaged();
   drawingview->Repair();
}

// AlignToGridChange stores the Selections' original positions.

AlignToGridChange::AlignToGridChange (Drawing* d, DrawingView* dv) :
ChangeNode(d, dv) {
   centerlist = drawing->GetCenter();
```

```
}

// Delete frees storage allocated for the list of original positions.

AlignToGridChange::~AlignToGridChange () {
  delete centerlist;
}

// Do aligns the Selections' lower left corners to the nearest grid
// point.

void AlignToGridChange::Do () {
  drawingview->EraseExcessHandles(oldsl);
  drawing->Select(oldsl);
  drawingview->Damaged();
  drawing->AlignToGrid();
  drawingview->Damaged();
  drawingview->Repair();
}

// Undo moves the Selections to their original positions.

void AlignToGridChange::Undo () {
  drawingview->EraseExcessHandles(oldsl);
  drawing->Select(oldsl);
  drawingview->Damaged();
  drawing->SetCenter(centerlist);
  drawingview->Damaged();
  drawingview->Repair();
}
```

listgroup.h

```
#ifndef listgroup_h
#define listgroup_h

#include "list.h"

// Declare imported types.

class PictSelection;
class SelectionList;

// A GroupNode contains a PictSelection pointer, a boolean value, and
// two SelectionList pointers.

class GroupNode : public BaseNode {
public:

  GroupNode(PictSelection*, boolean, SelectionList*);
  ~GroupNode();

  PictSelection* GetParent () { return parent; }
  boolean GetHasChildren () { return haschildren; }
  SelectionList* GetChildren () { return children; }
  SelectionList* GetChildrenGS () { return childrengs; }

protected:

  PictSelection* parent;// contains a Selection which may have children
  boolean haschildren;// true if this parent has children
  SelectionList* children;// lists the parent's children, if any
  SelectionList* childrengs;// stores the children's orig. graphic states

};

// A GroupList manages a list of GroupNodes.

class GroupList : public BaseList {
public:

  GroupNode* First();
  GroupNode* Last();
  GroupNode* Prev();
```

```
    GroupNode* Next();
    GroupNode* GetCur();
    GroupNode* Index(int);

};

// Cast these functions to return GroupNodes instead of BaseNodes.

inline GroupNode* GroupList::First () {
    return (GroupNode*) BaseList::First();
}

inline GroupNode* GroupList::Last () {
    return (GroupNode*) BaseList::Last();
}

inline GroupNode* GroupList::Prev () {
    return (GroupNode*) BaseList::Prev();
}

inline GroupNode* GroupList::Next () {
    return (GroupNode*) BaseList::Next();
}

inline GroupNode* GroupList::GetCur () {
    return (GroupNode*) BaseList::GetCur();
}

inline GroupNode* GroupList::Index (int index) {
    return (GroupNode*) BaseList::Index(index);
}

#endif
```

listgroup.c

```
#include "listgroup.h"
#include "listselectn.h"
#include "selection.h"

// GroupNode stores the parent, boolean value, children, and the
// children's original graphic states.

GroupNode::GroupNode (PictSelection* p, boolean h, SelectionList* sl) {
    parent = p;
    haschildren = h;
    children = new SelectionList;
    childrengs = new SelectionList;
    for (sl->First(); !sl->AtEnd(); sl->Next()) {
Selection* child = sl->GetCur()->GetSelection();
Selection* childgs = new Selection(child);
children->Append(new SelectionNode(child));
childrengs->Append(new SelectionNode(childgs));
    }
}

// Free storage allocated to list the children and allocated
// for their graphic states.

GroupNode::~GroupNode () {
    delete children;
    for (childrengs->First(); !childrengs->AtEnd(); childrengs->Next()) {
Selection* s = childrengs->GetCur()->GetSelection();
delete s;
    }
    delete childrengs;
}
```

listibrush.h

```
#ifndef listibrush_h
#define listibrush_h

#include "list.h"
```

```
// Declare imported types.

class IBrush;

// An IBrushNode contains an IBrush pointer.

class IBrushNode : public BaseNode {
public:

   IBrushNode (IBrush* b) { brush = b; }
   boolean SameValueAs (void* p) { return brush == p; }
   IBrush* GetBrush () { return brush; }

protected:

   IBrush* brush;// points to an IBrush

};

// An IBrushList manages a list of IBrushNodes.

class IBrushList : public BaseList {
public:

   IBrushNode* First();
   IBrushNode* Last();
   IBrushNode* Prev();
   IBrushNode* Next();
   IBrushNode* GetCur();
   IBrushNode* Index(int);

};

// Cast these functions to return IBrushNodes instead of BaseNodes.

inline IBrushNode* IBrushList::First () {
   return (IBrushNode*) BaseList::First();
}

inline IBrushNode* IBrushList::Last () {
   return (IBrushNode*) BaseList::Last();
}

inline IBrushNode* IBrushList::Prev () {
   return (IBrushNode*) BaseList::Prev();
}

inline IBrushNode* IBrushList::Next () {
   return (IBrushNode*) BaseList::Next();
}

inline IBrushNode* IBrushList::GetCur () {
   return (IBrushNode*) BaseList::GetCur();
}

inline IBrushNode* IBrushList::Index (int index) {
   return (IBrushNode*) BaseList::Index(index);
}

#endif
```

listicolor.h

```
#ifndef listicolor_h
#define listicolor_h

#include "list.h"

// Declare imported types.

class IColor;

// An IColorNode contains an IColor pointer.

class IColorNode : public BaseNode {
public:
```

```
    IColorNode (IColor* c) { color = c; }
    boolean SameValueAs (void* p) { return color == p; }
    IColor* GetColor () { return color; }

protected:

    IColor* color;// points to an IColor

};

// An IColorList manages a list of IColorNodes.

class IColorList : public BaseList {
public:

    IColorNode* First();
    IColorNode* Last();
    IColorNode* Prev();
    IColorNode* Next();
    IColorNode* GetCur();
    IColorNode* Index(int);

};

// Cast these functions to return IColorNodes instead of BaseNodes.

inline IColorNode* IColorList::First () {
    return (IColorNode*) BaseList::First();
}

inline IColorNode* IColorList::Last () {
    return (IColorNode*) BaseList::Last();
}

inline IColorNode* IColorList::Prev () {
    return (IColorNode*) BaseList::Prev();
}

inline IColorNode* IColorList::Next () {
    return (IColorNode*) BaseList::Next();
}

inline IColorNode* IColorList::GetCur () {
    return (IColorNode*) BaseList::GetCur();
}

inline IColorNode* IColorList::Index (int index) {
    return (IColorNode*) BaseList::Index(index);
}

#endif
```

listifont.h

```
#ifndef listifont_h
#define listifont_h

#include "list.h"

// Declare imported types.

class IFont;

// An IFontNode contains an IFont pointer.

class IFontNode : public BaseNode {
public:

    IFontNode (IFont* f) { font = f; }
    boolean SameValueAs (void* p) { return font == p; }
    IFont* GetFont () { return font; }

protected:
```

```
    IFont* font;// points to an IFont

};

// An IFontList manages a list of IFontNodes.

class IFontList : public BaseList {
public:

    IFontNode* First();
    IFontNode* Last();
    IFontNode* Prev();
    IFontNode* Next();
    IFontNode* GetCur();
    IFontNode* Index(int);

};

// Cast these functions to return IFontNodes instead of BaseNodes.

inline IFontNode* IFontList::First () {
    return (IFontNode*) BaseList::First();
}

inline IFontNode* IFontList::Last () {
    return (IFontNode*) BaseList::Last();
}

inline IFontNode* IFontList::Prev () {
    return (IFontNode*) BaseList::Prev();
}

inline IFontNode* IFontList::Next () {
    return (IFontNode*) BaseList::Next();
}

inline IFontNode* IFontList::GetCur () {
    return (IFontNode*) BaseList::GetCur();
}

inline IFontNode* IFontList::Index (int index) {
    return (IFontNode*) BaseList::Index(index);
}

#endif
```

listintrctr.h

```
#ifndef listintrctr_h
#define listintrctr_h

#include "list.h"

// Declare imported types.

class Interactor;

// An InteractorNode contains an Interactor pointer.

class InteractorNode : public BaseNode {
public:

    InteractorNode (Interactor* i) { interactor = i; }
    boolean SameValueAs (void* p) { return interactor == p; }
    Interactor* GetInteractor () { return interactor; }

protected:

    Interactor* interactor;// points to a Interactor

};

// An InteractorList manages a list of InteractorNodes.

class InteractorList : public BaseList {
public:
```

```
    InteractorNode* First();
    InteractorNode* Last();
    InteractorNode* Prev();
    InteractorNode* Next();
    InteractorNode* GetCur();
    InteractorNode* Index(int);

};

// Cast these functions to return InteractorNodes instead of
// BaseNodes.

inline InteractorNode* InteractorList::First () {
    return (InteractorNode*) BaseList::First();
}

inline InteractorNode* InteractorList::Last () {
    return (InteractorNode*) BaseList::Last();
}

inline InteractorNode* InteractorList::Prev () {
    return (InteractorNode*) BaseList::Prev();
}

inline InteractorNode* InteractorList::Next () {
    return (InteractorNode*) BaseList::Next();
}

inline InteractorNode* InteractorList::GetCur () {
    return (InteractorNode*) BaseList::GetCur();
}

inline InteractorNode* InteractorList::Index (int index) {
    return (InteractorNode*) BaseList::Index(index);
}

#endif
```

listipattern.h

```
#ifndef listipattern_h
#define listipattern_h

#include "list.h"

// Declare imported types.

class IPattern;

// An IPatternNode contains an IPattern pointer.

class IPatternNode : public BaseNode {
public:

    IPatternNode (IPattern* p) { pattern = p; }
    boolean SameValueAs (void* p) { return pattern == p; }
    IPattern* GetPattern () { return pattern; }

protected:

    IPattern* pattern;// points to an IPattern

};

// An IPatternList manages a list of IPatternNodes.

class IPatternList : public BaseList {
public:

    IPatternNode* First();
    IPatternNode* Last();
    IPatternNode* Prev();
    IPatternNode* Next();
    IPatternNode* GetCur();
    IPatternNode* Index(int);
```

```
};

// Cast these functions to return IPattemNodes instead of BaseNodes.

inline IPattemNode* IPattemList::First () {
   return (IPattemNode*) BaseList::First();
}

inline IPattemNode* IPattemList::Last () {
   return (IPattemNode*) BaseList::Last();
}

inline IPattemNode* IPattemList::Prev () {
   return (IPattemNode*) BaseList::Prev();
}

inline IPattemNode* IPattemList::Next () {
   return (IPattemNode*) BaseList::Next();
}

inline IPattemNode* IPattemList::GetCur () {
   return (IPattemNode*) BaseList::GetCur();
}

inline IPattemNode* IPattemList::Index (int index) {
   return (IPattemNode*) BaseList::Index(index);
}

#endif
```

listselectn.h

```
#ifndef listselectn_h
#define listselectn_h

#include "list.h"

// Declare imported types.

class Selection;

// A SelectionNode stores a pointer to a Selection.

class SelectionNode : public BaseNode {
public:

   SelectionNode (Selection* s) { selection = s; }
   boolean SameValueAs (void* p) { return selection == p; }
   Selection* GetSelection () { return selection; }

protected:

   Selection* selection;// points to a Selection

};

// A SelectionList manages a list of SelectionNodes.

class SelectionList : public BaseList {
public:

   SelectionNode* First();
   SelectionNode* Last();
   SelectionNode* Prev();
   SelectionNode* Next();
   SelectionNode* GetCur();
   SelectionNode* Index(int);

};

// Cast these functions to return SelectionNodes instead of BaseNodes.

inline SelectionNode* SelectionList::First () {
   return (SelectionNode*) BaseList::First();
```

```
}

inline SelectionNode* SelectionList::Last () {
   return (SelectionNode*) BaseList::Last();
}

inline SelectionNode* SelectionList::Prev () {
   return (SelectionNode*) BaseList::Prev();
}

inline SelectionNode* SelectionList::Next () {
   return (SelectionNode*) BaseList::Next();
}

inline SelectionNode* SelectionList::GetCur () {
   return (SelectionNode*) BaseList::GetCur();
}

inline SelectionNode* SelectionList::Index (int index) {
   return (SelectionNode*) BaseList::Index(index);
}

#endif
```

main.c

```
#include "idraw.h"
#include <InterViews/world.h>

// Predefine default properties for the window size, paint menus, and
// history.

static PropertyData properties[] = {
   { "*font1","*-courier-medium-r-*-80-*   Courier 8" },
   { "*font2","*-courier-medium-r-*-100-*  Courier 10" },
   { "*font3","*-courier-bold-r-*-120-*    Courier-Bold 12" },
   { "*font4","*-helvetica-medium-r-*-120-* Helvetica 12" },
   { "*font5","*-helvetica-medium-r-*-140-* Helvetica 14" },
   { "*font6","*-helvetica-bold-r-*-140-*  Helvetica-Bold 14" },
   { "*font7","*-helvetica-medium-o-*-140-* Helvetica-Oblique 14" },
   { "*font8","*-times-medium-r-*-120-*    Times-Roman 12" },
   { "*font9","*-times-medium-r-*-140-*    Times-Roman 14" },
   { "*font10","*-times-bold-r-*-140-*     Times-Bold 14" },
   { "*font11","*-times-medium-i-*-140-*   Times-Italic 14" },
{ "*brush1", "ffff 2 0 1" },
{ "*brush2", "ffff 1 0 1" },

// These brush types are not needed for a data flow diagram

/* ***** Start of Commented Out Code *****
   { "*brush1","none" },
   { "*brush2","ffff 1 0 0" },   removed all brush types except for
   { "*brush3","ffff 1 1 0" },   the type that has as its first end
   { "*brush4","ffff 1 0 1" },   endpoint the end without the arrow
   { "*brush5","ffff 1 1 1" },   head and it's second or last end
   { "*brush6","3333 1 0 0" },   point is the end with the arrowhead
   { "*brush7","3333 2 0 0" },
   { "*brush8","ffff 2 0 0" },
***** End of Comments Out Code ***** */

   { "*pattern1","none" },
   { "*pattern2","0.0" },
   { "*pattern3","1.0" },
   { "*arrowpattern1","none" },
   { "*arrowpattern2","0.0" },
   { "*arrowpattern3","1.0" },

// These patterns are not needed for a data flow diagram

/* ***** Start of Commented Out Code *****
   { "*pattern4","0.75" },
   { "*pattern5","0.5" },
   { "*pattern6","0.25" },
   { "*pattern7","1248" },
   { "*pattern8","8421" },
   { "*pattern9","f000" },
```

```
    { "*pattern10","8888" },
    { "*pattern11","f888" },
    { "*pattern12","8525" },
    { "*pattern13","cc33" },
    { "*pattern14","7bed" },
***** End of Commented Out Code ***** */

    { "*fgcolor1","Black" },
    { "*fgcolor2","Brown 42240 10752 10752" },
    { "*fgcolor3","Red" },
    { "*fgcolor4","Orange" },
    { "*fgcolor5","Yellow" },
    { "*fgcolor6","Green" },
    { "*fgcolor7","Blue" },
    { "*fgcolor8","Indigo 48896 0 65280" },
    { "*fgcolor9","Violet 20224 12032 20224" },
    { "*fgcolor10","White" },
    { "*fgcolor11","LtGray 50000 50000 50000" },
    { "*fgcolor12","DkGray 33000 33000 33000" },
    { "*bgcolor1","Black" },
    { "*bgcolor2","Brown 42240 10752 10752" },
    { "*bgcolor3","Red" },
    { "*bgcolor4","Orange" },
    { "*bgcolor5","Yellow" },
    { "*bgcolor6","Green" },
    { "*bgcolor7","Blue" },
    { "*bgcolor8","Indigo 48896 0 65280" },
    { "*bgcolor9","Violet 20224 12032 20224" },
    { "*bgcolor10","White" },
    { "*bgcolor11","LtGray 50000 50000 50000" },
    { "*bgcolor12","DkGray 33000 33000 33000" },
    { "*initialfont","2" },
    { "*initialbrush","1" },
    { "*initialpattern","3" },
    { "*initialarrowpattern", "2" },
    { "*initialfgcolor","1" },
    { "*initialbgcolor","10" },
    { "*history","20" },
    { "*reverseVideo","off" },
    { "*small","true" },
    { "*geometry", "+10+10"},
    { nil }
};

// Define window size options.

static OptionDesc options[] = {
    { "-l", "*small", OptionValueImplicit, "false" },
    { "-s", "*small", OptionValueImplicit, "true" },
    { nil }
};

// main creates a connection to the display server, creates idraw, and
// opens idraw's window.  After idraw stops running, main closes
// idraw's window, deletes everything it created, and returns success.

int main (int argc, char** argv) {
    World* world = new World("Idraw", properties, options, argc, argv);
    Idraw* idraw = new Idraw(argc, argv);

    world->InsertApplication(idraw);
    idraw->Run();
    world->Remove(idraw);

    delete idraw;
    delete world;

    const int SUCCESS = 0;
    return SUCCESS;
}
```

mapipaint.h

```
#ifndef mapipaint_h
#define mapipaint_h

#include <InterViews/defs.h>
```

```
// Declare imported types.

class BaseList;
class BaseNode;
class IBrush;
class IBrushList;
class IColor;
class IColorList;
class IFont;
class IFontList;
class IPattern;
class IPatternList;
class Interactor;

// A MapIPaint creates a list of predefined and user-defined entries.

class MapIPaint {
protected:

   void Init(BaseList*, Interactor*, const char*);
   void DefineEntries(BaseList*, Interactor*, const char*);
   void DefineInitial(Interactor*, const char*);
   virtual BaseNode* CreateEntry(const char*);

   int initial;// denotes which entry is used on startup

};

// A MapIBrush manages a list of predefined and user-defined brushes.

class MapIBrush : public MapIPaint {
public:

   MapIBrush(Interactor*, const char*);
   ~MapIBrush();

   int Size();
   boolean AtEnd();
   IBrush* First();
   IBrush* Last();
   IBrush* Prev();
   IBrush* Next();
   IBrush* GetCur();
   IBrush* Index(int);
   boolean Find(IBrush*);
   IBrush* GetInitial();

   IBrush* FindOrAppend(boolean, int, int, int, int);

protected:

   BaseNode* CreateEntry(const char*);

   IBrushList* ibrushlist;   // stores idraw's IBrushes

};

// A MapIColor manages a list of predefined and user-defined colors.

class MapIColor : public MapIPaint {
public:

   MapIColor(Interactor*, const char*);
   ~MapIColor();

   int Size();
   boolean AtEnd();
   IColor* First();
   IColor* Last();
   IColor* Prev();
   IColor* Next();
   IColor* GetCur();
   IColor* Index(int);
   boolean Find(IColor*);
   IColor* GetInitial();
```

```
    IColor* FindOrAppend(const char*, int, int, int);

protected:

    BaseNode* CreateEntry(const char*);

    IColorList* icolorlist;    // stores idraw's IColors

};

// A MapIFont manages a list of predefined and user-defined fonts.

class MapIFont : public MapIPaint {
public:

    MapIFont(Interactor*, const char*);
    ~MapIFont();

    int Size();
    boolean AtEnd();
    IFont* First();
    IFont* Last();
    IFont* Prev();
    IFont* Next();
    IFont* GetCur();
    IFont* Index(int);
    boolean Find(IFont*);
    IFont* GetInitial();

    IFont* FindOrAppend(const char*, const char*, const char*);

protected:

    BaseNode* CreateEntry(const char*);

    IFontList* ifontlist;    // stores idraw's IFonts

};

// A MapIPattern manages a list of predefined and user-defined
// patterns.

class MapIPattern : public MapIPaint {
public:

    MapIPattern(Interactor*, const char*);
    ~MapIPattern();

    int Size();
    boolean AtEnd();
    IPattern* First();
    IPattern* Last();
    IPattern* Prev();
    IPattern* Next();
    IPattern* GetCur();
    IPattern* Index(int);
    boolean Find(IPattern*);
    IPattern* GetInitial();

    IPattern* FindOrAppend(boolean, float, int*, int);

protected:

    BaseNode* CreateEntry(const char*);
    int CalcBitmap(float);

    void ExpandToFullSize(int*, int);

    IPatternList* ipatternlist;    // stores idraw's IPatterns

};

#endif
```

mapipaint.c

```
#include "ipaint.h"
#include "istring.h"
#include "listibrush.h"
#include "listicolor.h"
#include "listifont.h"
#include "listipattern.h"
#include "mapipaint.h"
#include <InterViews/interactor.h>
#include <bstring.h>
#include <stream.h>

// Init creates the list's entries, initializes the number denoting
// the initial entry, and ensures both list and initial are valid.

void MapIPaint::Init (BaseList* list, Interactor* i, const char* menu) {
    DefineEntries(list, i, menu);
    DefineInitial(i, menu);
    if (list->Size() == 0) {
fprintf(stderr, "No entries in %s menu, ", menu);
fprintf(stderr, "creating a default entry\n");
list->Append(CreateEntry(nil));
    }
    if (list->Index(initial-1) == nil) {
fprintf(stderr, "No attribute at %s entry %d, ", menu, initial);
fprintf(stderr, "setting initial attribute from first entry\n");
initial = 1;
    }
}

// DefineEntries reads predefined or user-defined attributes and
// creates entries in the list from these attributes. It retrieves
// each attribute using the concatenation of the menu's name and a
// number which increments from 1. The first undefined or empty
// attribute terminates the menu's definition.

void MapIPaint::DefineEntries (BaseList* list, Interactor* i,
const char* menu) {
    char menuproperty[20];
    int num = 1;
    sprintf(menuproperty, "%s%d", menu, num);

    const char* attribute = i->GetAttribute(menuproperty);
    while (attribute != nil && strlen(attribute) > 0) {
BaseNode* entry = CreateEntry(attribute);
if (entry != nil) {
    list->Append(entry);
} else {
    fprintf(stderr, "couldn't parse attribute for %s\n", menuproperty);
}

sprintf(menuproperty, "%s%d", menu, ++num);
attribute = i->GetAttribute(menuproperty);
    }
}

// DefineInitial reads a predefined or user-defined attribute to
// define the number denoting the initial entry. For example,
// "idraw.initialfont: 2" sets the initial font from the second entry.

void MapIPaint::DefineInitial (Interactor* i, const char* menu) {
    char property[20];
    sprintf(property, "initial%s", menu);

    const char* def = i->GetAttribute(property);
    char* definition = strdup(def); // some sscanfs write to their format...
    if (sscanf(definition, "%d", &initial) != 1) {
initial = 2;// default if we can't parse definition
fprintf(stderr, "can't parse attribute for %s, ", property);
fprintf(stderr, "value set to %d\n", initial);
    }
    delete definition;
}

// CreateEntry creates and returns an entry containing an attribute
// defined by the given string. A subclass must implement it.
```

```
BaseNode* MapIPaint::CreateEntry (const char*) {
  return new BaseNode;
}

// Skew comments/code ratio to work around cpp bug
;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;

// MapIBrush creates its brushes from predefined or user-defined
// attributes.

MapIBrush::MapIBrush (Interactor* i, const char* menu) {
  ibrushlist = new IBrushList;
  Init(ibrushlist, i, menu);
}

// ~MapIBrush frees storage allocated for the brushes and the list.

MapIBrush::~MapIBrush () {
  for (IBrush* brush = First(); !AtEnd(); brush = Next()) {
delete brush;
  }
  delete ibrushlist;
}

// Implement functions to give readonly access to the list.

int MapIBrush::Size () {
  return ibrushlist->Size();
}

boolean MapIBrush::AtEnd () {
  return ibrushlist->AtEnd();
}

IBrush* MapIBrush::First () {
  return ibrushlist->First()->GetBrush();
}

IBrush* MapIBrush::Last () {
  return ibrushlist->Last()->GetBrush();
}

IBrush* MapIBrush::Prev () {
  return ibrushlist->Prev()->GetBrush();
}

IBrush* MapIBrush::Next () {
  return ibrushlist->Next()->GetBrush();
}

IBrush* MapIBrush::GetCur () {
  return ibrushlist->GetCur()->GetBrush();
}

IBrush* MapIBrush::Index (int index) {
  return ibrushlist->Index(index)->GetBrush();
}

boolean MapIBrush::Find (IBrush* brush) {
  return ibrushlist->Find(brush);
}

IBrush* MapIBrush::GetInitial () {
  // remember index = 0-base, initial = 1-base
  return ibrushlist->Index(initial-1)->GetBrush();
}

// FindOrAppend checks if the list already has a brush with the same
// attributes as the given attributes. If so, it returns the brush
// already in the list; otherwise, it creates a new brush, appends it
// to the list, and returns it.

IBrush* MapIBrush::FindOrAppend (boolean none, int p, int w, int l, int r) {
  IBrush* brush = nil;
  if (none) {
```

```
    for (brush = First(); !AtEnd(); brush = Next()) {
        if (brush->None()) {
    return brush;
        }
    }
    brush = new IBrush;
        } else {
    for (brush = First(); !AtEnd(); brush = Next()) {
        if (!brush->None() &&
    brush->GetLinePattern() == p &&
    brush->Width() == w &&
    brush->LeftArrow() == l &&
    brush->RightArrow() == r)
        {
    return brush;
        }
    }
    brush = new IBrush(p, w, l, r);
        }

        ibrushlist->Append(new IBrushNode(brush));
        return brush;
    }

    // CreateEntry returns an entry containing a brush created by parsing
    // the given definition or an entry containing a default brush if no
    // definition was given. The definition usually contains two numbers
    // and two booleans: a 16-bit hexadecimal number to define the brush's
    // pattern, a decimal integer to define the brush's width in pixels,
    // either 0 or 1 to determine whether lines start from an arrowhead,
    // and either 0 or 1 to determine whether lines end in an arrowhead.
    // One definition may contain the string "none" to define the
    // nonexistent brush. I found out sscanf barfs if you put 0x before
    // the hexadecimal number so you can't do that.

    BaseNode* MapIBrush::CreateEntry (const char* def) {
        if (def == nil) {
    def = "ffff 1 0 0";
        }
        char* definition = strdup(def); // some sscanfs write to their format...

        BaseNode* entry = nil;
        int p, w;
        boolean l, r;
        boolean none = (definition[0] == 'n' || definition[0] == 'N');

        if (none) {
    entry = new IBrushNode(new IBrush);
        } else if (sscanf(definition, "%x %d %d %d", &p, &w, &l, &r) == 4) {
    entry = new IBrushNode(new IBrush(p, w, l, r));
        }
        delete definition;

        return entry;
    }

    // MapIColor creates its colors from predefined or user-defined
    // attributes.

    MapIColor::MapIColor (Interactor* i, const char* menu) {
        icolorlist = new IColorList;
        Init(icolorlist, i, menu);
    }

    // ~MapIColor frees storage allocated for the colors and the list.

    MapIColor::~MapIColor () {
        for (IColor* color = First(); !AtEnd(); color = Next()) {
    delete color;
        }
        delete icolorlist;
    }

    // Implement functions to give readonly access to the list.

    int MapIColor::Size () {
```

```
    return icolorlist->Size();
}

boolean MapIColor::AtEnd () {
    return icolorlist->AtEnd();
}

IColor* MapIColor::First () {
    return icolorlist->First()->GetColor();
}

IColor* MapIColor::Last () {
    return icolorlist->Last()->GetColor();
}

IColor* MapIColor::Prev () {
    return icolorlist->Prev()->GetColor();
}

IColor* MapIColor::Next () {
    return icolorlist->Next()->GetColor();
}

IColor* MapIColor::GetCur () {
    return icolorlist->GetCur()->GetColor();
}

IColor* MapIColor::Index (int index) {
    return icolorlist->Index(index)->GetColor();
}

boolean MapIColor::Find (IColor* color) {
    return icolorlist->Find(color);
}

IColor* MapIColor::GetInitial () {
    // remember index = 0-base, initial = 1-base
    return icolorlist->Index(initial-1)->GetColor();
}

// FindOrAppend checks if the list already has a color with the same
// name as the given name.  If so, it returns the color already in the
// list; otherwise, it creates a new color, appends it to the list,
// and returns it.

IColor* MapIColor::FindOrAppend (const char* name, int r, int g, int b) {
    IColor* color = nil;
    for (color = First(); !AtEnd(); color = Next()) {
if (strcmp(color->GetName(), name) == 0) {
    return color;
}
    }

    color = new IColor(name);
    if (!color->Valid()) {
delete color;
color = new IColor(r, g, b, name);
if (!color->Valid()) {
    fprintf(stderr, "invalid color name %s ", name);
    fprintf(stderr, "and intensities %d %d %d, ", r, g, b);
    fprintf(stderr, "substituting black\n");
    delete color;
    color = new IColor(black, name);
}
    }
    icolorlist->Append(new IColorNode(color));
    return color;
}

// CreateEntry returns an entry containing a color created by using
// the given name or an entry containing a default color if no name
// was given.

BaseNode* MapIColor::CreateEntry (const char* def) {
    if (def == nil) {
def = "Black";
```

```
    }
    char* definition = strdup(def); // some sscanfs write to their format...

    IColor* color = nil;
    char name[100];
    int r = 0, g = 0, b = 0;

    if (sscanf(definition, "%s %d %d %d", name, &r, &g, &b) == 4) {
  color = new IColor(r, g, b, name);
  if (!color->Valid()) {
    fprintf(stderr, "invalid intensities %d %d %d, ", r, g, b);
    fprintf(stderr, "using color name %s\n", name);
    delete color;
    color = new IColor(name);
    if (!color->Valid()) {
  fprintf(stderr, "invalid color name %s, ", name);
  fprintf(stderr, "substituting black\n");
  delete color;
  color = new IColor(black, name);
    }
  }
    } else if (sscanf(definition, "%s", name) == 1) {
  color = new IColor(name);
  if (!color->Valid()) {
    fprintf(stderr, "invalid color name %s, ", name);
    fprintf(stderr, "substituting black\n");
    delete color;
    color = new IColor(black, name);
  }
    }
    delete definition;

    BaseNode* entry = nil;
    if (color != nil) {
  entry = new IColorNode(color);
    }
    return entry;
}

// MapIFont creates its fonts from predefined or user-defined
// attributes.

MapIFont::MapIFont (Interactor* i, const char* menu) {
    ifontlist = new IFontList;
    Init(ifontlist, i, menu);
}

// ~MapIFont frees storage allocated for the fonts and the list.

MapIFont::~MapIFont () {
    for (IFont* font = First(); !AtEnd(); font = Next()) {
  delete font;
    }
    delete ifontlist;
}

// Implement functions to give readonly access to the list.

int MapIFont::Size () {
    return ifontlist->Size();
}

boolean MapIFont::AtEnd () {
    return ifontlist->AtEnd();
}

IFont* MapIFont::First () {
    return ifontlist->First()->GetFont();
}

IFont* MapIFont::Last () {
    return ifontlist->Last()->GetFont();
}

IFont* MapIFont::Prev () {
    return ifontlist->Prev()->GetFont();
```

```
}

IFont* MapIFont::Next () {
   return ifontlist->Next()->GetFont();
}

IFont* MapIFont::GetCur () {
   return ifontlist->GetCur()->GetFont();
}

IFont* MapIFont::Index (int index) {
   return ifontlist->Index(index)->GetFont();
}

boolean MapIFont::Find (IFont* font) {
   return ifontlist->Find(font);
}

IFont* MapIFont::GetInitial () {
   // remember index = 0-base, initial = 1-base
   return ifontlist->Index(initial-1)->GetFont();
}

// FindOrAppend checks if the list already has a font with the same
// attributes as the given attributes (except for the name because it
// depends on the window system used). If so, it returns the font
// already in the list; otherwise, it creates a new font, appends it
// to the list, and returns it.

IFont* MapIFont::FindOrAppend (const char* name, const char* pf,
const char* ps) {
   IFont* font = nil;
   for (font = First(); !AtEnd(); font = Next()) {
if (strcmp(font->GetPrintFont(), pf) == 0
   && strcmp(font->GetPrintSize(), ps) == 0)
{
   return font;
}
   }

   font = new IFont(name, pf, ps);
   if (!font->Valid()) {
fprintf(stderr, "invalid font name %s, ", name);
fprintf(stderr, "substituting stdfont\n");
delete font;
font = new IFont("stdfont", pf, ps);
   }
   ifontlist->Append(new IFontNode(font));
   return font;
}

// CreateEntry returns an entry containing a font created by parsing
// the given definition or an entry containing a default font if no
// definition was given. The definition must contain three strings
// separated by whitespace to define a font. The first string defines
// the font's name, the second string the corresponding print font,
// and the third string the print size.

BaseNode* MapIFont::CreateEntry (const char* def) {
   if (def == nil) {
def = "stdfont Courier 10";
   }
   char* definition = strdup(def); // some sscanfs write to their format...

   BaseNode* entry = nil;
   char name[100];
   char pf[100];
   char ps[10];

   if (sscanf(definition, "%s %s %s", name, pf, ps) == 3) {
IFont* font = new IFont(name, pf, ps);
if (!font->Valid()) {
   fprintf(stderr, "invalid font name %s, ", name);
   fprintf(stderr, "substituting stdfont\n");
   delete font;
   font = new IFont("stdfont", pf, ps);
```

```
}
entry = new IFontNode(font);
    }
    delete definition;

    return entry;
}

// MapIPattern creates its patterns from predefined or user-defined
// attributes.

MapIPattern::MapIPattern (Interactor* i, const char* menu) {
    ipatternlist = new IPatternList;
    Init(ipatternlist, i, menu);
}

// ~MapIPattern frees storage allocated for the patterns and the list.

MapIPattern::~MapIPattern () {
    for (IPattern* pattern = First(); !AtEnd(); pattern = Next()) {
delete pattern;
    }
    delete ipatternlist;
}

// Implement functions to give readonly access to the list.

int MapIPattern::Size () {
    return ipatternlist->Size();
}

boolean MapIPattern::AtEnd () {
    return ipatternlist->AtEnd();
}

IPattern* MapIPattern::First () {
    return ipatternlist->First()->GetPattern();
}

IPattern* MapIPattern::Last () {
    return ipatternlist->Last()->GetPattern();
}

IPattern* MapIPattern::Prev () {
    return ipatternlist->Prev()->GetPattern();
}

IPattern* MapIPattern::Next () {
    return ipatternlist->Next()->GetPattern();
}

IPattern* MapIPattern::GetCur () {
    return ipatternlist->GetCur()->GetPattern();
}

IPattern* MapIPattern::Index (int index) {
    return ipatternlist->Index(index)->GetPattern();
}

boolean MapIPattern::Find (IPattern* pattern) {
    return ipatternlist->Find(pattern);
}

IPattern* MapIPattern::GetInitial () {
    // remember index = 0-base, initial = 1-base
    return ipatternlist->Index(initial-1)->GetPattern();
}

// Skew comments/code ratio to work around cpp bug
;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;

// FindOrAppend checks if the list already has a pattern with the same
// attributes as the given attributes. If so, it returns the pattern
// already in the list; otherwise, it creates a new pattern, appends
// it to the list, and returns it.
```

```
IPattern* MapIPattern::FindOrAppend (boolean none, float graylevel,
int data[patternHeight], int size) {
   IPattern* pattern = nil;
   if (none) {
for (pattern = First(); !AtEnd(); pattern = Next()) {
   if (pattern->None()) {
return pattern;
   }
}
pattern = new IPattern;
   } else if (graylevel != -1) {
for (pattern = First(); !AtEnd(); pattern = Next()) {
   if (pattern->GetGrayLevel() == graylevel) {
return pattern;
   }
}
int shade = CalcBitmap(graylevel);
pattern = new IPattern(shade, graylevel);
   } else {
ExpandToFullSize(data, size);
for (pattern = First(); !AtEnd(); pattern = Next()) {
   if (pattern->GetSize() != 0) {
const int* cmpdata = pattern->GetData();
if (bcmp(data, cmpdata, patternHeight * sizeof(int)) == 0) {
   return pattern;
}
   }
}
pattern = new IPattern(data, size);
   }

   ipatternlist->Append(new IPatternNode(pattern));
   return pattern;
}

// CreateEntry returns an entry containing a pattern created by
// parsing the given definition or an entry containing a default
// pattern if no definition was given. The definition usually
// contains either one or patternHeight 16-bit hexadecimal numbers to
// define either a 4x4 pattern or a patternWidth x patternHeight
// pattern. One definition may contain the string "none" to define
// the nonexistent pattern. I found out sscanf barfs if you put 0x
// before the hexadecimal number so you can't do that.

BaseNode* MapIPattern::CreateEntry (const char* def) {
   if (def == nil) {
def = "0.0";
   }
   char* definition = strdup(def); // some sscanfs write to their format...

   BaseNode* entry = nil;
   boolean none = (definition[0] == 'n' || definition[0] == 'N');

   if (none) {
entry = new IPatternNode(new IPattern);
   } else {
if (strchr(definition, '.') != nil) {
   float graylevel;

   if (sscanf(definition, "%f", &graylevel) == 1) {
int shade = CalcBitmap(graylevel);
entry = new IPatternNode(new IPattern(shade, graylevel));
   }
} else {
   istream from(strlen(definition) + 1, definition);
   char buffer[80];
   int data[patternHeight];

   for (int i = 0; from >> buffer && i < patternHeight; i++) {
if (sscanf(buffer, "%x", &data[i]) != 1) {
   break;
}
   }

   if (i == 1 || i == 8 || i == patternHeight) {
ExpandToFullSize(data, i);
```

```
entry = new IPatternNode(new IPattern(data, i));
    }
}
    }
    delete definition;

    return entry;
}

// CalcBitmap calculates a 4x4 bitmap to represent a shade having the
// given gray level.

int MapIPattern::CalcBitmap (float graylevel) {
    static const int SHADES = 17;
    static int shades[SHADES] = {
0xffff, 0xefff, 0xefbf, 0xafbf,
0xafaf, 0xadaf, 0xada7, 0xa5a7,
0xa5a5, 0x85a5, 0x8525, 0x0525,
0x0505, 0x0405, 0x0401, 0x0001,
0x0000
    };
    return shades[round(graylevel * (SHADES - 1))];
}

// ExpandToFullSize expands the bitmap from 4x4 or 8x8 to 16x16.

void MapIPattern::ExpandToFullSize (int data[patternHeight], int size) {
    if (size == 1) {
int seed = data[0];
for (int i = 0; i < 4; i++) {
    data[i] = (seed & 0xf000) >> 12;
    data[i] |= data[i] << 4;
    data[i] |= data[i] << 8;
    data[i+4] = data[i];
    data[i+8] = data[i];
    data[i+12] = data[i];
    seed <<= 4;
}
    } else if (size == 8) {
for (int i = 0; i < 8; i++) {
    data[i] &= 0xff;
    data[i] |= data[i] << 8;
    data[i+8] = data[i];
}
    } else if (size == patternHeight) {
const unsigned int patternWidthMask = ~(~0 << patternWidth);
for (int i = 0; i < patternHeight; i++) {
    data[i] &= patternWidthMask;
}
    } else {
fprintf(stderr, "invalid size passed to ExpandToFullSize\n");
    }
}
```

mapkey.h

```
#ifndef mapkey_h
#define mapkey_h

#include <InterViews/defs.h>

// Declare imported types.

class Interactor;

// A MapKey maps characters to Interactors.

static const int MAXCHAR = 127;// Maximum value of any character.

class MapKey {
public:

    MapKey();

    void Enter(Interactor*, char);
```

```
    Interactor* LookUp(char);
    const char* ToStr(char);

protected:

    Interactor* array[MAXCHAR + 1]; // stores Interactors by character

};

#endif
```

mapkey.c

```
#include "mapkey.h"
#include <ctype.h>
#include <InterViews/Std/stdio.h>

// MapKey clears MapKey's array to all nils.

MapKey::MapKey () {
    for (int i = 0; i <= MAXCHAR; i++) {
array[i] = nil;
    }
}

// Enter enters an Interactor into a slot in MapKey's array.  If the
// slot doesn't exist or another Interactor already occupies it,
// Enter prints a warning message.

void MapKey::Enter (Interactor* i, char c) {
    if (c >= 0 && c <= MAXCHAR) {
if (array[c] == nil) {
    array[c] = i;
} else {
    fprintf(stderr, "MapKey: slot %d already occupied!\n", c);
}
    } else {
fprintf(stderr, "MapKey: slot %d not in array!\n", c);
    }
}

// LookUp returns the Interactor associated with the given character
// or nil if there's no Interactor or the character's out of bounds.

Interactor* MapKey::LookUp (char c) {
    if (c >= 0 && c <= MAXCHAR) {
return array[c];
    } else {
fprintf(stderr, "MapKey: slot %d not in array!\n", c);
return nil;
    }
}

// ToStr returns a printable string representing the given character.
// The caller must copy the returned string because the next call of
// ToStr will change it.

const char* MapKey::ToStr (char c) {
    static char key[3];
    if (!isprint(c)) {
c = toascii(c + 0100);
key[0] = '^';
    } else {
key[0] = ' ';
    }
    key[1] = c;
    key[2] = 0;
    return key;
}
```

page.h

```
#ifndef page_h
```

```
#define page_h

#include <InterViews/Graphic/picture.h>

// Declare imported types.

class PictSelection;

// A Page draws a grid, picture, and boundary. It can constrain
// points to lie only on the grid.

static const int GRID_DEFAULTSPACING = 8;

class Page : public Picture {
public:

   Page(double w, double h, double b, Graphic* = nil);
   ~Page();

   Color* GetBackgroundColor();
   boolean GetGridGravity();
   double GetGridSpacing();
   boolean GetGridVisibility();
   PictSelection* GetPicture();

   void SetGridGravity(boolean);
   void SetGridSpacing(double);
   void SetGridVisibility(boolean);
   void SetPicture(PictSelection*);

   void Center(float, float, float);
   void Constrain(Coord&, Coord&);
   void ToggleOrientation();

protected:

   void getExtent(float&, float&, float&, float&, float&, Graphic*);

   void draw(Canvas*, Graphic*);
   void drawGrid(Canvas*, Graphic*);
   void drawBoundary(Canvas*, Graphic*);

   void drawClipped(Canvas*, Coord, Coord, Coord, Coord, Graphic*);
   void drawGridClipped(Canvas*, Coord, Coord, Coord, Coord, Graphic*);
   void drawPictureClipped(Canvas*, Coord, Coord, Coord, Coord, Graphic*);
   void drawBoundaryClipped(Canvas*, Coord, Coord, Coord, Coord, Graphic*);

   int DefinePoints(Coord, Coord, Coord, Coord, Graphic*);

   double pagewidth;// stores exact width of page
   double pageheight;// stores exact height of page
   int pgwidth;// stores integral width of page
   int pgheight;// stores integral height of page
   PBrush* border;// stores brush for drawing boundary
   boolean gravity;// will constrain points to grid if true
   boolean visibility;// will draw grid points if true
   double spacing_pixels;// stores spacing in units of pixels
   double spacing_points;// stores spacing in units of printer's points
   Coord* x, *y;// stores grid points
   Graphic* grid_gs;// stores attributes for drawing grid/boundary
   Transformer* grid_tt;// stores matrix for drawing grid/boundary
   PictSelection* picture;// stores picture

};

// Define access functions to get and set members' values.

inline boolean Page::GetGridGravity () {
   return gravity;
}

inline double Page::GetGridSpacing () {
   return spacing_points;
}

inline boolean Page::GetGridVisibility () {
```

```
    return visibility;
}

inline PictSelection* Page::GetPicture () {
    return picture;
}

inline void Page::SetGridGravity (boolean g) {
    gravity = g;
}

inline void Page::SetGridVisibility (boolean v) {
    visibility = v;
}

#endif
```

page.c

```
#define _POSIX_SOURCE
#include "ipaint.h"
#include "page.h"
#include "slpict.h"
#include <InterViews/Std/math.h>
#include <InterViews/Graphic/lines.h>
#include <InterViews/Graphic/polygons.h>
#include <InterViews/Graphic/util.h>
#include <InterViews/canvas.h>
#include <InterViews/painter.h>
#include <InterViews/transformer.h>

// Page starts out with gravity disabled, spacing set to the default
// value, visibility disabled, and an empty picture.

Page::Page (double w, double h, double b, Graphic* gs) : (gs) {
    pagewidth = w;
    pageheight = h;
    pgwidth = round(w);
    pgheight = round(h);
    border = new PBrush(0xffff, round(b));
    gravity = false;
    spacing_pixels = 0.0;
    spacing_points = 0.0;
    visibility = false;
    x = y = nil;
    SetGridSpacing(GRID_DEFAULTSPACING);
    grid_gs = new FullGraphic;
    grid_tt = new Transformer;
    grid_gs->SetBrush(psingle);
    grid_gs->SetColors(pblack, pwhite);
    grid_gs->SetPattern(psolid);
    grid_gs->SetTransformer(grid_tt);
    grid_gs->FillBg(true);
    picture = nil;
    SetPicture(nil);
    if (GetTransformer() == nil) {
SetTransformer(new Transformer);
    }
}

Page::~Page () {
    delete border;
    delete x;
    delete y;
//    delete grid_gs;
}

// GetBackgroundColor gets the background color for DrawingView.

Color* Page::GetBackgroundColor () {
    PColor* bg = grid_gs->GetBgColor();
    return *bg;
}
```

```
// SetPicture replaces the old picture with a new picture (creating an
// empty one if necessary) and deletes the old picture.

void Page::SetPicture (PictSelection* newpic) {
  if (newpic == nil) {
newpic = new PictSelection;
  }
  if (picture != nil) {
Remove(picture);
delete picture;
  }
  picture = newpic;
  Append(picture);
}

// SetGridSpacing changes the spacing between the grid points, which
// also requires reallocating the number of points in the grid.

void Page::SetGridSpacing (double p) {
  if (spacing_points != p) {
spacing_points = p;
spacing_pixels = spacing_points * points;
delete x;
delete y;
int x_count = int(pagewidth/spacing_pixels) + 1;
int y_count = int(pageheight/spacing_pixels) + 1;
int count = x_count * y_count;
x = new Coord[count];
y = new Coord[count];
  }
}

// Center replaces the page's matrix with a freshly generated matrix
// to center the page in the window and get rid of accumulated
// roundoff errors.

void Page::Center (float mag, float wincx, float wincy) {
  Transformer* t = GetTransformer();
  float cx, cy;

  if (t->Rotated90()) {
*t = identity;
ToggleOrientation();
  } else {
*t = identity;
  }
  GetCenter(cx, cy);
  Scale(mag, mag, cx, cy);
  Translate(wincx - cx, wincy - cy);
}

// Constrain replaces the given point with the closest grid point if
// gravity has been enabled.

void Page::Constrain (Coord& x, Coord& y) {
  if (gravity) {
float x0, y0;
grid_tt->InvTransform(float(x), float(y), x0, y0);
x0 = round(round(x0/spacing_pixels) * spacing_pixels);
y0 = round(round(y0/spacing_pixels) * spacing_pixels);
grid_tt->Transform(x0, y0, x0, y0);
x = round(x0);
y = round(y0);
  }
}

// ToggleOrientation examines the picture's matrix to see what state
// it is in and flips the matrix to the other state.

void Page::ToggleOrientation () {
  Transformer* t = GetTransformer();
  float l, b, dx, dy;

  if (t->Rotated90()) {
t->Transform(0.0, -pagewidth, dx, dy);
t->Transform(0.0, 0.0, l, b);
```

```
Translate(dx - l, dy - b);
Rotate(90.0, l, b);
    } else {
t->Transform(0.0, 0.0, l, b);
t->Transform(0.0, pagewidth, dx, dy);
Rotate(-90.0, l, b);
Translate(dx - l, dy - b);
    }
}

// getExtent returns the page's dimensions as its extent instead of
// returning the extent of its picture so idraw's panner will always
// show the page's extent, not the picture's extent.

void Page::getExtent (float& l, float& b, float& cx, float& cy,
float& tol, Graphic* gs) {
    float dummy1, dummy2;
    transformRect(0.0, 0.0, pagewidth, pageheight, l, b, dummy1, dummy2, gs);
    transform(pagewidth/2, pageheight/2, cx, cy, gs);
    tol = 0;
}

// draw draws the grid, picture, and boundary.

void Page::draw (Canvas* c, Graphic* gs) {
    concatTransformer(nil, gs->GetTransformer(), grid_tt);
    drawGrid(c, grid_gs);
    Picture::draw(c, gs);
    drawBoundary(c, grid_gs);
}

// drawGrid passes the work off to drawGridClipped.

void Page::drawGrid (Canvas* c, Graphic* gs) {
    Coord xmax = c->Width();
    Coord ymax = c->Height();
    drawGridClipped(c, 0, 0, xmax, ymax, gs);
}

// drawBoundary draws the boundary around the page.

void Page::drawBoundary (Canvas* c, Graphic* gs) {
    gs->SetBrush(border);
    update(gs);
    pRect(c, 0, 0, pgwidth, pgheight);
}

// drawClipped draws part of the grid, picture, and boundary.

void Page::drawClipped (
    Canvas* c, Coord l, Coord b, Coord r, Coord t, Graphic* gs
) {
    concatTransformer(nil, gs->GetTransformer(), grid_tt);
    drawGridClipped(c, l, b, r, t, grid_gs);
    drawPictureClipped(c, l, b, r, t, gs);
    drawBoundaryClipped(c, l, b, r, t, grid_gs);
}

// drawGridClipped grids the given area with points if visibility has
// been enabled. It sends the points in chunks of MAXCHUNK points
// each because Xlib does not yet break up too-big requests.

void Page::drawGridClipped (Canvas* c, Coord l, Coord b, Coord r, Coord t,
Graphic* gs) {
    if (visibility) {
gs->SetBrush(psingle);
update(gs);
int ntotal = DefinePoints(l, b, r, t, gs);
int nsent = 0;
while (nsent < ntotal) {
    const int MAXCHUNK = 6000;
    int nchunk = min(MAXCHUNK, ntotal - nsent);
    pMultiPoint(c, &x[nsent], &y[nsent], nchunk);
    nsent += nchunk;
}
    }
```

```
}

// drawPictureClipped goes right ahead and draws the picture, letting
// it decide if its extent intersects the clipping box.

void Page::drawPictureClipped (
    Canvas* c, Coord l, Coord b, Coord r, Coord t, Graphic* gs
) {
    register RefList* i;
    Graphic* gr;
    FullGraphic gstemp;
    Transformer ttemp;

    gstemp.SetTransformer(&ttemp);
    for (i = refList->First(); i != refList->End(); i = i->Next()) {
gr = getGraphic(i);
concatGraphic(gr, gr, gs, &gstemp);
drawClippedGraphic(gr, c, l, b, r, t, &gstemp);
    }
    gstemp.SetTransformer(nil); /* to avoid deleting ttemp explicitly */
}

// drawBoundaryClipped draws part of the boundary around the page.

void Page::drawBoundaryClipped (
    Canvas* c, Coord l, Coord b, Coord r, Coord t, Graphic* gs
) {
    BoxObj clipBox(l, b, r, t);

    Coord x0, y0, x1, y1, x2, y2, x3, y3;
    transform(0, 0, x0, y0, gs);
    transform(0, pgheight, x1, y1, gs);
    transform(pgwidth, 0, x2, y2, gs);
    transform(pgwidth, pgheight, x3, y3, gs);
    LineObj lLine(x0, y0, x1, y1);
    LineObj bLine(x0, y0, x2, y2);
    LineObj rLine(x2, y2, x3, y3);
    LineObj tLine(x1, y1, x3, y3);

    gs->SetBrush(border);
    update(gs);
    if (clipBox.Intersects(lLine)) {
pLine(c, 0, 0, 0, pgheight);
    }
    if (clipBox.Intersects(bLine)) {
pLine(c, 0, 0, pgwidth, 0);
    }
    if (clipBox.Intersects(rLine)) {
pLine(c, pgwidth, 0, pgwidth, pgheight);
    }
    if (clipBox.Intersects(tLine)) {
pLine(c, 0, pgheight, pgwidth, pgheight);
    }
}

// DefinePoints defines just enough points to grid the given area.

int Page::DefinePoints (Coord l, Coord b, Coord r, Coord t, Graphic* gs) {
    float x0, y0, x1, y1;
    invTransformRect(float(l),float(b),float(r),float(t), x0, y0, x1, y1, gs);
    x0 = round(x0/spacing_pixels) * spacing_pixels;
    y0 = round(y0/spacing_pixels) * spacing_pixels;
    x0 = max(float(0.0), x0);
    y0 = max(float(0.0), y0);
    x1 = min(x1, (float)pagewidth);
    y1 = min(y1, (float)pageheight);
    int x_count = int((x1 - x0)/spacing_pixels) + 1;
    int y_count = int((y1 - y0)/spacing_pixels) + 1;
    int count = x_count * y_count;

    for (int i = 0; i < x_count; i++) {
Coord ix = round(x0 + i*spacing_pixels);
for (int j = i; j < count; j += x_count) {
    x[j] = ix;
}
    }
```

```
    for (i = 0; i < y_count; i++) {
Coord iy = round(y0 + i*spacing_pixels);
for (int j = i * x_count; j < (i+1) * x_count; j++) {
    y[j] = iy;
}
    }

    return count;
}
```

panel.h

```
#ifndef panel_h
#define panel_h

#include "highlighter.h"

// Declare imported and used-before-defined types.

class Editor;
class PanelItem;

// A Panel displays several items but highlights only one item.

class Panel : public HighlighterParent {
public:

    Panel();

    void Enter(PanelItem*, char);
    PanelItem* LookUp(char);

    void SetCur(PanelItem*);
    PanelItem* GetCur();

    void PerformCurrentFunction(Event&);
    void PerformTemporaryFunction(Event&, char);

protected:

    PanelItem* cur;// stores currently selected item
    PanelItem* items[128];// stores items by their associated character

};

// A PanelItem displays two text labels and performs a function.

class PanelItem : public Highlighter {
public:

    PanelItem(Panel*, const char*, const char*, char, Editor*);
    ~PanelItem();

    void Handle(Event&);

    virtual void Perform(Event&);
virtual void SetMessage();

protected:

    void Reconfig();
    void Redraw(Coord, Coord, Coord, Coord);
    void Resize();

    Panel* panel;// stores panel which this item belongs to
    char* name;// stores item's text label
    char* key;// stores label of key which selects this item
    char* msg;      // stores message shown in message block

    Coord name_x, name_y;// stores position at which to display name
    Coord key_x, key_y;// stores position at which to display key
    Coord side;// size of largest square fitting in canvas
    Coord offx;// horizontal offset needed to center square
    Coord offy;// vertical offset needed to center square
```

```
Editor* editor; // stores editor to be used in setting message
};

#endif
```

panel.c

```
#include "dfd_defs.h"
#include "editor.h"
#include "istring.h"
#include "panel.h"
#include <InterViews/event.h>
#include <InterViews/font.h>
#include <InterViews/painter.h>
#include <InterViews/sensor.h>
#include <InterViews/shape.h>
#include <ctype.h>
#include <InterViews/Std/stdio.h>

// Define the maximum value of any character.

static const int MAXCHAR = 127;

// Panel starts with no currently highlighted or stored items.

Panel::Panel () {
    cur = nil;
    for (int i = 0; i <= MAXCHAR; i++) {
items[i] = nil;
    }
}

// Enter stores an item using its associated character as its index
// and tells it it can get its highlight painter from us.

void Panel::Enter (PanelItem* i, char c) {
    if (c >= 0 && c <= MAXCHAR) {
items[c] = i;
    }
    i->SetHighlighterParent(this);
}

// LookUp returns the item associated with the given character.

PanelItem* Panel::LookUp (char c) {
    PanelItem* i = nil;
    if (c >= 0 && c <= MAXCHAR) {
i = items[c];
    }
    return i;
}

// SetCur sets the currently highlighted item.  The panel highlights
// only one item at any time.

void Panel::SetCur (PanelItem* item) {
    if (cur != nil) {
cur->Highlight(false);
    }
    cur = item;
    if (cur != nil) {
cur->Highlight(true);
    }
}

// GetCur returns the currently highlighted item.

PanelItem* Panel::GetCur () {
    return cur;
}

// PerformCurrentFunction tells the currently highlighted item to
// perform its function.

void Panel::PerformCurrentFunction (Event& e) {
    if (cur != nil) {
```

```
cur->Perform(e);
    }
}

// PerformTemporaryFunction temporarily selects the item associated
// with the given character and tells it to perform its function.

void Panel::PerformTemporaryFunction (Event& e, char c) {
    PanelItem* tmp = LookUp(c);
    if (tmp != nil) {
      PanelItem* prev = cur;
      tmp->SetMessage();
      SetCur(tmp);
      tmp->Perform(e);
  if (prev != nil) {
       prev->SetMessage();
      SetCur(prev);
      }
    }
}

// PanelItem stores the panel it belongs to and its text labels.

PanelItem::PanelItem (Panel* p, const char* l, const char* k, char c,
Editor* e) {
    p->Enter(this, c);
    panel = p;
editor = e;
msg = new char [MAXMSGLEN+1];
    name = strdup(l ? l : "");
    key = strdup(k ? k : "");
    input = updownEvents;
    input->Reference();
}

// Free storage allocated to store the text labels.

PanelItem::~PanelItem () {
    delete name;
    delete key;
}

// Handle highlights the item if the user clicks on it or types its
// associated character (mapped by the program).

void PanelItem::Handle (Event& e) {
    switch (e.eventType) {
    case DownEvent:
    case KeyEvent:
SetMessage();
panel->SetCur(this);
break;
    default:
break;
    }
}

// sets the value of the message in the message block

void PanelItem::SetMessage() {
// define it in your subclass
}

// Perform performs the item's function.

void PanelItem::Perform (Event&) {
    // define it in your subclass
}

// Reconfig pads the item's shape to accomodate its text labels.
// Basing padding on the font in use ensures the padding will change
// proportionally with changes in the font's size.

static const float WIDTHPAD = 1.5; // fraction of font->Width(EM)
static const float HTPAD = 0.77;   // fraction of font->Height()
static const float KEYPAD = 1./6.; // fraction of font->Width(EM)
```

```
static const char* EM = "m";   // widest alphabetic character in any font

void PanelItem::Reconfig () {
    Highlighter::Reconfig();
    Font* font = output->GetFont();
    int xpad = round(WIDTHPAD * font->Width(EM));
    int ypad = round(HTPAD * font->Height());
    shape->width = font->Width(name) + (2 * xpad);
    shape->height = font->Height() + (2 * ypad);
    shape->Rigid(2 * xpad, hfil, 2 * ypad, 0);
}

// Redraw displays the text labels.

void PanelItem::Redraw (Coord l, Coord b, Coord r, Coord t) {
    output->ClearRect(canvas, l, b, r, t);
    output->FillBg(false);
    output->Text(canvas, name, name_x, name_y);
    output->Text(canvas, key, key_x, key_y);
    output->FillBg(true);
}

// Resize calculates the text labels' positions.  For the convenience
// of subclasses which want to draw graphic labels centered in the
// canvas and as large as possible while maintaining a 1:1 aspect
// ratio, Resize also calculates side, offx, and offy which define the
// largest possible square centered in the canvas.

void PanelItem::Resize () {
    Font* font = output->GetFont();
    name_x = max(0, (xmax - font->Width(name) + 1) / 2);
    name_y = (ymax - font->Height() + 1) / 2;
    int pad = round(KEYPAD * font->Width(EM));
    key_x = xmax - font->Width(key) - pad;
    key_y = pad;
    side = min(xmax, ymax);
    offx = (xmax - side) / 2;
    offy = (ymax - side) / 2;
}
```

pdmenu.h

```
#ifndef pdmenu_h
#define pdmenu_h

#include "highlighter.h"

// Declare imported and used-before-defined types.

class PullDownMenuButton;

class PullDownMenuActivator;
class PullDownMenuCommand;

// A PullDownMenuBar displays several activators and coordinates which
// activator will open its menu.

class PullDownMenuBar : public HighlighterParent {
public:

    PullDownMenuBar();
    ~PullDownMenuBar();
    void Enter(PullDownMenuButton*);

    boolean Contains(Interactor*);

    boolean MenuActive();

    boolean MenuShouldActivate(PullDownMenuButton*);

    void MenuActivate(PullDownMenuButton*);
    void MenuDeactivate();

protected:
```

```
    void GrowButtons();

    PullDownMenuButton* cur;// stores currently active activator
    int sizebuttons;// stores current size of dynamic array
    int numbuttons;// stores number of activators in array
    PullDownMenuButton**
buttons;// stores bar's interior activators

};


class PullDownMenuButton : public Highlighter {
public:

    PullDownMenuButton(PullDownMenuBar*, const char*);
    PullDownMenuButton(PullDownMenuActivator*, const char*);
    ~PullDownMenuButton();

    virtual boolean Contains(Interactor *) { return false; }
    virtual void Open() { Highlight(true); }
    virtual void Close() { Highlight(false); }

    virtual void Execute(Event&) { }

protected:
    PullDownMenuBar* barParent;// stores bar containing this activator
    PullDownMenuActivator* buttonParent;

    virtual void Reconfig();
    virtual void Redraw(Coord, Coord, Coord, Coord);
    virtual void Resize();

    char* name;// stores activator's text label
    Coord name_x, name_y;// stores position at which to display name
};


// A PullDownMenuActivator displays a text label and opens a menu when
// you activate it.

class PullDownMenuActivator : public PullDownMenuButton {
public:

    PullDownMenuActivator(PullDownMenuBar*, const char*);
    PullDownMenuActivator(PullDownMenuActivator*, const char*);
    ~PullDownMenuActivator();

    void Enter(PullDownMenuButton*);
    boolean Contains(Interactor *child);

    void SetMenu(Scene*);
    void Handle(Event&);

    void Open();
    void Close();
    void ChainClose();

protected:
    void HandleBar(Event&);
    void HandleButton(Event&);

    boolean IsOpen;

    Scene* menu;// stores menu to be opened when activated
    void GrowButtons();

    int sizebuttons;// stores current size of dynamic array
    int numbuttons;// stores number of commands in array
    PullDownMenuButton**
buttons;// stores activator's interior commands


};

// A PullDownMenuCommand displays a text label and executes a command.
```

```
class PullDownMenuCommand : public PullDownMenuButton {
public:

   PullDownMenuCommand(PullDownMenuActivator*, const char*, const char*);
   PullDownMenuCommand(PullDownMenuBar*, const char*, const char*);
   ~PullDownMenuCommand();

//********* change by Eagle 15 Sep 94, made this a virtual function
   virtual void Handle(Event&);

   virtual void Execute(Event&);

protected:

   void Reconfig();
   void Redraw(Coord, Coord, Coord, Coord);
   void Resize();

   PullDownMenuActivator*
activator;// stores activator which this cmd belongs to
   char* key;// stores label of key which selects this cmd

   Coord key_x, key_y;// stores position at which to display key

};

// A PullDownMenuDivider displays a horizontal line extending the full
// width of the menu, dividing it into two submenus.

class PullDownMenuDivider : public PullDownMenuCommand {
public:

   PullDownMenuDivider();

protected:

   void Redraw(Coord, Coord, Coord, Coord);

};

#endif
```

pdmenu.c

```
#include "istring.h"
#include "pdmenu.h"
#include <InterViews/event.h>
#include <InterViews/font.h>
#include <InterViews/frame.h>
#include <InterViews/painter.h>
#include <InterViews/sensor.h>
#include <InterViews/shape.h>
#include <InterViews/world.h>
#include <bstring.h>

/* Define the initial number of elements to allocate space for. */

static const int INITIALSIZE = 15;

/*
 * PullDownMenuBar starts with no currently active or stored
 * activators although it allocates some initial space to store them.
 */

PullDownMenuBar::PullDownMenuBar () : cur(0) {
   sizebuttons = INITIALSIZE;
   numbuttons = 0;
   buttons = new PullDownMenuButton*[sizebuttons];
}

/*
 * Free storage allocated for the dynamic array.
 */

PullDownMenuBar::~PullDownMenuBar () {
   delete [] buttons;
```

```
}

/*
 * Enter stores an interior activator and tells it it can get its
 * highlight painter from us.
 */

void PullDownMenuBar::Enter (PullDownMenuButton* act) {
  if (numbuttons == sizebuttons) {
GrowButtons();
  }
  buttons[numbuttons++] = act;
}

/*
 * Contains returns true if the child is one of the bar's activators
 * or one of the activators' commands.
 */

boolean PullDownMenuBar::Contains (Interactor* child) {
  for (int i = 0; i < numbuttons; i++) {
if ((buttons[i] == child) || (buttons[i]->Contains(child))) {
  return true;
}
  }
  return false;
}

/*
 * MenuActive returns true if any of the bar's activators have opened
 * a menu.
 */

boolean PullDownMenuBar::MenuActive () {
  return cur ? 1 : 0;
}

/*
 * MenuShouldActivate returns true if any of the bar's activators
 * EXCEPT the given one have opened a menu.
 */

boolean PullDownMenuBar::MenuShouldActivate (PullDownMenuButton *btn) {
  return cur && cur != btn;
}

/*
 * MenuActivate highlights the given activator, opens the activator's
 * menu, and stores it as the currently active activator.
 */

void PullDownMenuBar::MenuActivate (PullDownMenuButton* act) {
  act->Highlight(true);
  cur = act;
  act->Open();
}

/*
 * MenuDeactivate closes the currently active activator's menu,
 * unhighlights the activator, and stores no more active activators.
 */

void PullDownMenuBar::MenuDeactivate () {
  cur->Close();
  cur->Highlight(false);
  cur = 0;
}

/*
 * GrowActivators increases the dynamic array's size to make room for
 * more activators to be stored.
 */

void PullDownMenuBar::GrowButtons () {
  PullDownMenuButton** oldacts = buttons;
  sizebuttons += INITIALSIZE/2;
```

```
    buttons = new PullDownMenuButton*[sizebuttons];
    bcopy(oldacts, buttons, numbuttons * sizeof(PullDownMenuButton*));
    delete oldacts;
  }

  PullDownMenuButton::PullDownMenuButton (PullDownMenuActivator* b,
  const char* n) :
      barParent(b->barParent), buttonParent(b) {
    name = strdup(n ? n : "");
    input = new Sensor(onoffEvents);
    input->CatchButton(DownEvent, LEFTMOUSE);
    input->CatchButton(UpEvent, LEFTMOUSE);
    SetHighlighterParent(barParent);
  }

  PullDownMenuButton::PullDownMenuButton (PullDownMenuBar* b, const char* n) :
      barParent(b), buttonParent(0) {
    name = strdup(n ? n : "");
    input = new Sensor(onoffEvents);
    input->CatchButton(DownEvent, LEFTMOUSE);
    input->CatchButton(UpEvent, LEFTMOUSE);
    SetHighlighterParent(barParent);
  }

  /*
   * Free storage allocated for members.
   */

  PullDownMenuButton::~PullDownMenuButton () {
    delete name;
  }

  /*
   * Reconfig pads the activator's shape to accomodate its text label.
   * Basing padding on the font in use ensures the padding will change
   * proportionally with changes in the font's size.
   */

  static const float WIDTHPAD = 1.0;/* fraction of font->Width(EM) */
  static const float CMDHTPAD = 0.1;/* fraction of font->Height() */
  static const float ACTHTPAD = 0.2;/* fraction of font->Height() */
  static const char* EM = "m";  /* widest alphabetic character in any font */

  void PullDownMenuButton::Reconfig () {
    Highlighter::Reconfig();
    Font* font = output->GetFont();
    int xpad = round(WIDTHPAD * font->Width(EM));
    int ypad = round(ACTHTPAD * font->Height());
    shape->width = font->Width(name) + (2 * xpad);
    shape->height = font->Height() + (2 * ypad);
    shape->Rigid(shape->width - xpad, 0, 2 * ypad, 0);
  }

  /*
   * Redraw displays the text label.
   */

  void PullDownMenuButton::Redraw (Coord l, Coord b, Coord r, Coord t) {
    output->ClearRect(canvas, l, b, r, t);
    output->Text(canvas, name, name_x, name_y);
  }

  /*
   * Resize calculates the text label's position.
   */

  void PullDownMenuButton::Resize () {
    Font* font = output->GetFont();
    name_x = max(0, (xmax - font->Width(name) + 1) / 2);
    name_y = (ymax - font->Height() + 1) / 2;
  }


  /*
   * PullDownMenuActivator stores the bar it belongs to and its text
   * label.  It starts off with an empty menu and no stored commands
```

```
 * although it allocates some initial space to store the commands.  It
 * must catch the same mouse button PullDownMenuCommand catches.
 */

PullDownMenuActivator::PullDownMenuActivator (PullDownMenuBar* b,
const char* n)  : PullDownMenuButton(b,n), IsOpen(false) {
   b->Enter(this);
   menu = new ShadowFrame;
   menu->SetCanvasType(CanvasSaveUnder);
   sizebuttons = INITIALSIZE;
   numbuttons = 0;
   buttons = new PullDownMenuButton*[sizebuttons];
}

PullDownMenuActivator::PullDownMenuActivator (PullDownMenuActivator* b,
const char* n)  : PullDownMenuButton(b,n), IsOpen(false) {
   b->Enter(this);
   menu = new ShadowFrame;
   menu->SetCanvasType(CanvasSaveUnder);
   sizebuttons = INITIALSIZE;
   numbuttons = 0;
   buttons = new PullDownMenuButton*[sizebuttons];
}

/*
 * Free storage allocated for members.
 */

PullDownMenuActivator::~PullDownMenuActivator () {
   delete buttons;
   delete menu;
}

/*
 * SetMenu sets the menu's actual contents.
 */

void PullDownMenuActivator::SetMenu (Scene* box) {
   menu->Insert(box);
}

void PullDownMenuActivator::ChainClose() {
  Close();
  if (buttonParent)
    buttonParent->ChainClose();
  else if (barParent->MenuActive())
    barParent->MenuDeactivate();
}

/*
 * Handle works together with the bar to determine whether any
 * activator has opened or should open a menu and accordingly tells
 * the bar to deactivate or activate an activator.
 */
void PullDownMenuActivator::Handle(Event &e) {
  if (buttonParent)
    HandleButton(e);
  else
    HandleBar(e);
}

void PullDownMenuActivator::HandleButton (Event& e) {
   switch (e.eventType) {
   case UpEvent:
      ChainClose();
break;
   case OnEvent:
Open();
break;
   case OffEvent:
while (1) {
  Read(e);
        if (e.eventType == OnEvent)
          if ((e.target != this) && !(Contains(e.target))) {
 Close();
            break;
```

```
            }
    if (e.eventType == UpEvent) {
      ChainClose();
            break;
    }
          e.target->Handle(e);
  }
       e.target->Handle(e);
 break;
    }
 }

 void PullDownMenuActivator::HandleBar (Event& e) {
    switch (e.eventType) {
    case DownEvent:
      if (!barParent->MenuActive()) {
        barParent->MenuActivate(this);
        while (e.eventType != UpEvent) {
          Read(e);
     if (e.eventType == UpEvent)
        barParent->MenuDeactivate();
     if (barParent->Contains(e.target))
        e.target->Handle(e);
  }
       }
 break;
    case UpEvent:
 break;
    case OnEvent:
      if (barParent->MenuShouldActivate(this)) {
        barParent->MenuDeactivate();
     barParent->MenuActivate(this);
 }
       Highlight(true);
 break;
    case OffEvent:
      if (!IsOpen)
        Highlight(false);
 break;
    }
 }

 /*
  * Enter stores an interior command and tells it it can get its
  * highlight painter from our bar like us too.
  */

 void PullDownMenuActivator::Enter (PullDownMenuButton* cmd) {
    if (numbuttons == sizebuttons) {
 GrowButtons();
    }
    buttons[numbuttons++] = cmd;
    cmd->SetHighlighterParent(barParent);
 }

 /*
  * Contains returns true if the command is one of the activator's
  * commands.
  */

 boolean PullDownMenuActivator::Contains (Interactor* cmd) {
    for (int i = 0; i < numbuttons; i++) {
 if ((buttons[i] == cmd) || (buttons[i]->Contains(cmd))){
    return true;
 }
    }
    return false;
 }

 /*
  * Open inserts the menu into the scene with the menu's top left
  * corner aligned with the activator's bottom left corner.
  */

 void PullDownMenuActivator::Open () {
    PullDownMenuButton::Open();
```

```
    if (IsOpen) return;
    IsOpen = true;
    World* world = GetWorld();
    Coord x = 0;
    Coord y = 0;
    if (buttonParent) {
      x = shape->width;
      y = shape->height;
    }
    GetRelative(x, y, world);
    world->InsertPopup(menu, x, y, TopLeft);
}

/*
 * Close makes the activator's menu disappear.
 */

void PullDownMenuActivator::Close() {
    PullDownMenuButton::Close();
    if (IsOpen) {
      for (int i = 0; i < numbuttons; i++)
   buttons[i]->Close();
      menu->Parent()->Remove(menu);
    }
    IsOpen = false;
}

/*
 * GrowCommands increases the dynamic array's size to make room for
 * more commands to be stored.
 */

void PullDownMenuActivator::GrowButtons() {
    PullDownMenuButton** oldcmds = buttons;
    sizebuttons += INITIALSIZE/2;
    buttons = new PullDownMenuButton*[sizebuttons];
    bcopy(oldcmds, buttons, numbuttons * sizeof(PullDownMenuButton*));
    delete oldcmds;
}

/*
 * PullDownMenuCommand stores the activator it belongs to and its text
 * labels.  It catches only one mouse button to prevent the user from
 * accidentally executing a command upon another button's release.
 */

PullDownMenuCommand::PullDownMenuCommand (PullDownMenuActivator* b,
const char* n, const char* k) : PullDownMenuButton(b,n) {
    if (b != nil) {
b->Enter(this);
    }
    key = strdup(k ? k : "");
}

PullDownMenuCommand::PullDownMenuCommand (PullDownMenuBar* b,
const char* n, const char* k) : PullDownMenuButton(b,n) {
    if (b != nil) {
b->Enter(this);
    }
    key = strdup(k ? k : "");
}

PullDownMenuCommand::~PullDownMenuCommand () {
    delete key;
}

/*
 * Highlight or unhighlight the command when the mouse passes
 * over and execute the command when the user releases the button
 * types its associated character (mapped by the program).
 */

void PullDownMenuCommand::Handle (Event& e) {
    switch (e.eventType) {
    case OnEvent:
      if (!buttonParent) {
```

```
    if (!barParent->MenuActive())
      Highlight(true);
  }
  else
      Highlight(true);
  break;
    case OffEvent:
  Highlight(false);
  break;
    case UpEvent:
  if (!buttonParent) {
    if (!barParent->MenuActive())
      Execute(e);
  }
  else
      Execute(e);
  Highlight(false);
  if (buttonParent)
    buttonParent->ChainClose();
  break;
    case KeyEvent:
  //activator->Flush();
  Execute(e);
  break;
    case DownEvent:
  break;
    }
  }

  /*
   * Execute carries out the command's purpose.
   */

  void PullDownMenuCommand::Execute (Event&) {
    /* define it in your subclass */
  }

  /*
   * Reconfig pads the command's shape to accomodate its text labels.
   */

  void PullDownMenuCommand::Reconfig () {
    Highlighter::Reconfig();
    Font* font = output->GetFont();
    int xpad = round(WIDTHPAD * font->Width(EM));
    int ypad = round(CMDHTPAD * font->Height());
    shape->width = font->Width(name) + (2 * xpad);
    shape->width += font->Width(key) + (2 * xpad);
    shape->height = font->Height() + (2 * ypad);
    shape->Rigid(0, hfil, 0, 0);
  }

  /*
   * Redraw displays the text labels.
   */

  void PullDownMenuCommand::Redraw (Coord l, Coord b, Coord r, Coord t) {
    PullDownMenuButton::Redraw(l,b,r,t);
    output->Text(canvas, key, key_x, key_y);
  }

  /*
   * Resize calculates the text labels' positions.
   */

  void PullDownMenuCommand::Resize () {
    Font* font = output->GetFont();
    int xpad = round(WIDTHPAD * font->Width(EM));
    name_x = xpad;
    name_y = (ymax - font->Height() + 1) / 2;
    key_x = xmax - font->Width(key) - xpad;
    key_y = name_y;
  }

  /*
   * PullDownMenuDivider listens to no events so it will neither
```

```
 * highlight itself nor execute any command.
 */

PullDownMenuDivider::PullDownMenuDivider () :
    PullDownMenuCommand((PullDownMenuBar *) nil, nil, nil) {
  Listen(noEvents);
}

/*
 * Redraw displays a horizontal line spanning the canvas.
 */

void PullDownMenuDivider::Redraw (Coord l, Coord b, Coord r, Coord t) {
  output->ClearRect(canvas, l, b, r, t);
  Coord hy = ymax / 2;
  output->Line(canvas, l, hy, r, hy);
}
```

psdlcomp.h

```
/*
 * This file contains the interafce for the following objects
 * GraphComponent, Vertex, Operator, Terminator, Edge
 * These are the components of a PSDL graph.  Each object
 * contains the idraw components necessary to create that
 * object in the editors drawing.  GraphComponent serves as
 * the base class from which Vertex and Edge are derived directly.
 * Operator and Terminator are further derived from Vertex
 *
 * This file was created specifically for the CAPS graphic editor.
 *
 * Created by:   C.S.Eagle
 * Last changes:  November 9, 1994
 *
 */

#ifndef __psdlcomp_h
#define __psdlcomp_h

#include <InterViews/Std/stdio.h>

class Edge;
class EdgeList;
class Operator;
class TextBuffer;
class TextSelection;
class EllipseSelection;
class BSplineSelection;
class RectSelection;
class Selection;

class GraphComponent {
public:
  GraphComponent(Selection *,TextSelection* = 0);
  ~GraphComponent();
  void SetLabel(TextSelection*);
  TextSelection *GetLabel();
  char *GetLabelString();
  char *GetValidLabelString();
  int GetClassId();
  virtual void SetShape(Selection *s);
  Selection *GetShape() {return shape;}

protected:
  Selection *shape;
  TextSelection *label;
  GraphComponent *me;
};

class Vertex : public GraphComponent {
public:
  Vertex(Selection*,TextSelection* = 0);
  ~Vertex();
  void SetId(int newId) {id = newId;}
  int GetId() {return id;}
  TextSelection *GetMET();
```

```
    char *GetMETString();
    void SetTag(TextSelection*);
    TextSelection* GetTag() {return tagSel;};
    EdgeList *GetInputs() {return inputs;}
    EdgeList *GetOutputs() {return outputs;}
    void AddInput(Edge *);
    void AddOutput(Edge *);
    void AddState(Edge *) {};
    void AddTrigger_If(char*) {}
    void AddTrigger_All(char*) {}
    void AddTrigger_Some(char*) {}
    void AddTriggerTo(char*) {}
    void GenerateAda(FILE*);
    boolean HasTriggerByAll(char*);
    void WritePSDL(FILE*);
    boolean HasConstraints() {return hasConstraints;}
    void WritePSDLConstraints(FILE*);

  protected:
    void WriteProcSpec(FILE*,const char*);
    TextSelection *met;
    TextSelection *tagSel;
    EdgeList *inputs;
    EdgeList *outputs;
    int id;
    boolean hasConstraints;
  };

class Operator : public Vertex {
  public:
    Operator(EllipseSelection*,TextSelection* = 0,
          TextSelection* = 0);
    ~Operator();
    EllipseSelection *GetEllipse();
    void SetMET(TextSelection*);
};

class Terminator : public Vertex {
  public:
    Terminator(RectSelection*,TextSelection* = 0);
    ~Terminator();
    RectSelection *GetRect();
  };

#define STATE_EDGE 0
#define SAMPLED_STREAM 1
#define DATAFLOW_STREAM 2

class Edge : public GraphComponent {
  public:
    Edge(BSplineSelection*, Vertex* = 0, Vertex* = 0,
       TextSelection* = 0,TextSelection* = 0);
    ~Edge();
    BSplineSelection *GetBSpline();
    void SetLatency(TextSelection*);
    TextSelection *GetLatency();
    char *GetLatencyString();
    Vertex *GetFromOp() {return fromOp;}
    char *GetFromVertexLabel();
    Vertex *GetToOp() {return toOp;}
    char *GetToVertexLabel();
    boolean IsState() {return state;}
    boolean IsStream() {return !state;}
    int StreamType();
    void SetDataType(char*);
    char *GetDataType();
    void WriteType(FILE*);

protected:
    TextSelection *latency;
    TextSelection *tagSel;
    Vertex *fromOp;
    Vertex *toOp;
    char *dataType;
    boolean state;
};
```

```
#endif
```

psdlcomp.c

```
/*
 * This file contains the implementation for the following objects
 * GraphComponent, Vertex, Operator, Terminator, Edge
 * These are the components of a PSDL graph.  Each object
 * contains the idraw components necessary to create that
 * object in the editors drawing.  GraphComponent serves as
 * the base class from which Vertex and Edge are derived directly.
 * Operator and Terminator are further derived from Vertex
 *
 * This file was created specifically for the CAPS graphic editor.
 *
 * Created by:   C.S.Eagle
 * Last changes:  November 9, 1994
 *
 */

#include "dfdclasses.h"
#include "dfd_defs.h"
#include "istring.h"
#include "list.h"
#include "psdlcomp.h"
#include "selection.h"
#include "sltext.h"
#include "slellipses.h"
#include "slpolygons.h"
#include "slsplines.h"
#include "psdllists.h"
#include <InterViews/textbuffer.h>


// GraphComponent constructor.  me is for casting to subclasses

GraphComponent::GraphComponent(Selection *s, TextSelection *ls) {
  me = this;
  SetLabel(ls);
  SetShape(s);
}


// GraphComponent destructor.  leave shape up to subclasses

GraphComponent::~GraphComponent() {
  delete label;
}


// set the components label

void GraphComponent::SetLabel(TextSelection *ls) {
  label = ls;
  if (ls)
    label->SetOwner(me);
}


// set the components shape

void GraphComponent::SetShape(Selection *s) {
  shape = s;
  if (s)
    shape->SetOwner(me);
}


// retrieve the components label

TextSelection *GraphComponent::GetLabel() {
  return label;
}
```

```
// retrieve the components label string

char *GraphComponent::GetLabelString() {
  return label ? label->GetString() : 0;
}


// retrieve the label string with invalid chars removed
// for use in ADA and PSDL texts

char *GraphComponent::GetValidLabelString() {
  return label ? label->GetValidString() : 0;
}


// retrieve the components class id

int GraphComponent::GetClassId() {
  return shape->GetClassId();
}


// vertex constructor

Vertex::Vertex(Selection *s, TextSelection *ls) : GraphComponent(s) {
  me = this;
  SetShape(s);
  SetLabel(ls);
  inputs = new EdgeList;
  outputs = new EdgeList;
  tagSel = 0;
  id = 0;
  met = 0;
  hasConstraints = 0;
}


// Vertex destructor

Vertex::~Vertex() {
  delete inputs;
  delete outputs;
  delete met;
}


// Set decomposition tag

void Vertex::SetTag(TextSelection *ts) {
  tagSel = ts;
  if (ts)
    tagSel->SetOwner(me);
}


// retrieve the Vertex's MET

TextSelection *Vertex::GetMET() {
  return met;
}


// retrieve the MET string

char *Vertex::GetMETString() {
  return met ? met->GetString() : 0;
}


// Add an Edge to the Vertex's input list

void Vertex::AddInput(Edge *e) {
  inputs->Append(new EdgeNode(e));
}
```

```
// add an edge to the vertex's output list

void Vertex::AddOutput(Edge *e) {
  outputs->Append(new EdgeNode(e));
}


// write an ADA procedure specification for the Vertex

void Vertex::WriteProcSpec(FILE *fptr, const char *pname) {
  int hasparm = 0;
  fprintf(fptr,"\tprocedure %s",pname);
  for (inputs->First(); !inputs->AtEnd();inputs->Next()) {
    Edge *ids = inputs->GetCur()->GetEdge();
    char *iparm = ids->GetValidLabelString();
    if (hasparm) {
      if (iparm)
        fprintf(fptr,";\n\t\t%s\t: UNDEFINED_TYPE",iparm);
      else
        fprintf(fptr,";\n\t\tUNDEFINED_NAME\t: UNDEFINED_TYPE");
    }
    else {
      hasparm = 1;
      if (iparm)
        fprintf(fptr,"(%s\t: UNDEFINED_TYPE",iparm);
      else
        fprintf(fptr,"(UNDEFINED_NAME\t: UNDEFINED_TYPE");
    }
    delete [] iparm;
  }
  for (outputs->First(); !outputs->AtEnd();outputs->Next()) {
   Edge *ods = outputs->GetCur()->GetEdge();
    char *oparm = ods->GetValidLabelString();
    if (hasparm) {
      if (oparm)
        fprintf(fptr,";\n\t\t%s\t: out UNDEFINED_TYPE",oparm);
      else
        fprintf(fptr,";\n\t\tUNDEFINED_NAME\t: out UNDEFINED_TYPE");
    }
    else {
      hasparm = 1;
      if (oparm)
        fprintf(fptr,"(%s\t: out UNDEFINED_TYPE",oparm);
      else
        fprintf(fptr,"(UNDEFINED_NAME\t: out UNDEFINED_TYPE");
    }
    delete [] oparm;
  }
  if (hasparm)
    fprintf(fptr,")");
}


// generate ADA shell code for the vertex

void Vertex::GenerateAda(FILE *fptr) {
  char *txt = GetValidLabelString();
  fprintf(fptr,"package %s_PKG is\n\n", txt);
  WriteProcSpec(fptr,txt);
  fprintf(fptr,";\n\nend %s_PKG;\n\n",txt);
  fprintf(fptr,"package body %s_PKG is\n\n",txt);
  WriteProcSpec(fptr,txt);
  fprintf(fptr," is\n\n");
  fprintf(fptr,"\tbegin\n\t\tnull;\n\tend %s;\n\n",txt);
  fprintf(fptr,"end %s_PKG;",txt);
  delete txt;
}


// add a trigger by all constraint to the vertex

boolean Vertex::HasTriggerByAll(char *id) {
  if (id) {
    return false;
  }
```

```
  return false;
}


// write the PSDL specification for the vertex

void Vertex::WritePSDL(FILE *fptr) {
  char *idstr = GetValidLabelString();
  fprintf(fptr,"%s%s\n%s",OPER_TKN,idstr,SPEC_TKN);
  delete [] idstr;
  inputs->WriteStreamTypes(fptr,INPUT_TKN);
  outputs->WriteStreamTypes(fptr,OUTPUT_TKN);
  inputs->WriteStateTypes(fptr," STATES\n");
  if (GetClassId() == OPERATOR) {
    if (met) {
      char *metstr = GetMETString();
      fprintf(fptr,"%s%s\n",MET_TKN,metstr);
      delete [] metstr;
    }
  }
  else
    fprintf(fptr,"%s 0\n",MET_TKN);
  fprintf(fptr,"%s",END_TKN);
}


// write the PSDL constraints for the vertex

void Vertex::WritePSDLConstraints(FILE *fptr) {
  if (hasConstraints) {
    char *name = label ? GetValidLabelString() : 0;
    if (label)
      fprintf(fptr," %s%s\n",OPER_TKN,label);
    else
      fprintf(fptr," %s",CON_OP_TKN);
  }
}


// Operator constructor

Operator::Operator(EllipseSelection *e, TextSelection *ls, TextSelection *m) :
    Vertex(e) {
  me = this;
  SetShape(e);
  SetLabel(ls);
  SetMET(m);
}


// Operator destructor.  shape handled here

Operator::~Operator() {
  delete (EllipseSelection *) shape;
}

void Operator::SetMET(TextSelection *m) {
  met = m;
  if (m)
    met->SetOwner(me);
}


// retrieve the Operators Ellipse shape

EllipseSelection *Operator::GetEllipse() {
  return (EllipseSelection *) shape;
}


// Terminator constructor

Terminator::Terminator(RectSelection *r, TextSelection *ls) :
    Vertex(r) {
  me = this;
  SetShape(r);
```

```
  SetLabel(ls);
}

// Terminator destructor.  shape handled here

Terminator::~Terminator() {
  delete (RectSelection *) shape;
}


// retrieve the Terminators Rect shape

RectSelection *Terminator::GetRect() {
  return (RectSelection *) shape;
}


// Edge constructor

Edge::Edge(BSplineSelection *b, Vertex *from, Vertex *to,
    TextSelection *ls, TextSelection *lat) :
    GraphComponent(b), fromOp(from), toOp(to) {

  me = this;
  dataType = 0;
  state = (to == from) ? 1 : 0;
  SetShape(b);
  SetLabel(ls);
  SetLatency(lat);
  if (!state) {
    b->SetStream();
    b->SetClassId(DATAFLOW_SPLINE);
  }
  else
    b->SetClassId(SELFLOOP);
}


// Edge destructor.  shape handled here

Edge::~Edge() {
  delete (BSplineSelection *) shape;
  delete latency;
}


// retrieve the Edges BSpline shape

BSplineSelection *Edge::GetBSpline() {
  return (BSplineSelection *) shape;
}


// set the edges latency

void Edge::SetLatency(TextSelection *l) {
  if (state) return;
  latency = l;
  if (l)
    l->SetOwner(me);
}


// retrieve the edges latency

TextSelection *Edge::GetLatency() {
  return latency;
}


// retrieve the string value for the latency

char *Edge::GetLatencyString() {
  return latency ? latency->GetString() : 0;
}
```

```
// get the label of the from vertex

char *Edge::GetFromVertexLabel() {
  return fromOp ? fromOp->GetValidLabelString() : 0;
}


// get the label of the to vertex

char *Edge::GetToVertexLabel() {
  return toOp ? toOp->GetValidLabelString() : 0;
}


// return the stream type of the edge

int Edge::StreamType() {
  if (state)
    return STATE_EDGE;
  if (toOp) {
    char *id = GetValidLabelString();
    if (toOp->HasTriggerByAll(id))
      return DATAFLOW_STREAM;
  }
  return SAMPLED_STREAM;
}


// set the data type for the edge

void Edge::SetDataType(char *dt) {
  if (dataType)
    delete [] dataType;
  if (dt) {
    dataType = new char[strlen(dt) + 1];
    strcpy(dataType,dt);
  }
}


// return a copy of the edges dataType

char *Edge::GetDataType() {
  if (dataType) {
    char *temp = new char[strlen(dataType)+1];
    strcpy(temp,dataType);
    return temp;
  }
  return 0;
}


// output the label and type as a type declaration

void Edge::WriteType(FILE *fptr) {
  char *idstr = label ? label->GetValidString() : ID_TKN;
  if (dataType)
    fprintf(fptr,"   %s : %s",idstr,dataType);
  else
    fprintf(fptr,"   %s : UNDEFINED_TYPE",idstr);
  delete [] idstr;
}
```

psdllists.h

```
/*
 * This file contains the interafce for the following objects
 * VertexNode, VertexList, EdgeNode, EdgeList
 * These derive from the idraw components BaseNode and BaseList
 * and are used to keef track of the underlying dataflow diagram
 * represented by the current graph
 *
 * This file was created specifically for the CAPS graphic editor.
 *
```

```
 * Created by:   C.S.Eagle
 * Last changes:  November 9, 1994
 *
 */

#ifndef __psdllists_h
#define __psdllists_h

#include "list.h"
#include <InterViews/defs.h>
#include <InterViews/Std/stdio.h>

class BSplineSelection;
class Edge;
class EllipseSelection;
class Operator;
class TextSelection;
class Selection;
class Vertex;


class VertexNode : public BaseNode {
 public:
  VertexNode(Vertex *v) : vx(v) { }
  Vertex *GetVertex() { return vx; }
 protected:
  Vertex *vx;
};

class VertexList : public BaseList {
public:
  VertexList();
  void Append(VertexNode *);
  VertexNode* First();
  VertexNode* Last();
  VertexNode* Prev();
  VertexNode* Next();
  VertexNode* GetCur();
  VertexNode* Index(int);

  void SetCur(Vertex*);

  Vertex* FindOp(Coord, Coord);
  Vertex* FindOp(int);

  void Replace(Selection*, Selection *);
  void Remove(Selection*);

protected:
  int currentId;
};

inline VertexNode* VertexList::First() {
  return (VertexNode*) BaseList::First();
}

inline VertexNode* VertexList::Last() {
  return (VertexNode*) BaseList::Last();
}

inline VertexNode* VertexList::Prev() {
  return (VertexNode*) BaseList::Prev();
}

inline VertexNode* VertexList::Next() {
  return (VertexNode*) BaseList::Next();
}

inline VertexNode* VertexList::GetCur() {
  return (VertexNode*) BaseList::GetCur();
}

inline VertexNode* VertexList::Index(int index) {
  return (VertexNode*) BaseList::Index(index);
}
```

```
class EdgeNode : public BaseNode {
 public:
  EdgeNode(Edge *v) : e(v) { }
  Edge *GetEdge() {return e;}
 protected:
  Edge *e;
};

class EdgeList : public BaseList {
public:
  EdgeNode* First();
  EdgeNode* Last();
  EdgeNode* Prev();
  EdgeNode* Next();
  EdgeNode* GetCur();
  EdgeNode* Index(int);

  void SetCur(Edge*);

  void Replace(Selection*, Selection*);
  void Remove(Selection*);

  void WriteStreamTypes(FILE*,char*);
  void WriteStateTypes(FILE*,char*);

};

inline EdgeNode* EdgeList::First() {
  return (EdgeNode*) BaseList::First();
}

inline EdgeNode* EdgeList::Last() {
  return (EdgeNode*) BaseList::Last();
}

inline EdgeNode* EdgeList::Prev() {
  return (EdgeNode*) BaseList::Prev();
}

inline EdgeNode* EdgeList::Next() {
  return (EdgeNode*) BaseList::Next();
}

inline EdgeNode* EdgeList::GetCur() {
  return (EdgeNode*) BaseList::GetCur();
}

inline EdgeNode* EdgeList::Index(int index) {
  return (EdgeNode*) BaseList::Index(index);
}

#endif
```

psdllists.c

```
/*
 * This file contains the implementation for the following objects
 * VertexNode, VertexList, EdgeNode, EdgeList
 * These derive from the idraw components BaseNode and BaseList
 * and are used to keef track of the underlying dataflow diagram
 * represented by the current graph
 *
 * This file was created specifically for the CAPS graphic editor.
 *
 * Created by:   C.S.Eagle
 * Last changes:  November 9, 1994
 *
 */

#include "dfdclasses.h"
#include "psdllists.h"
#include "psdlcomp.h"
#include "sltext.h"
#include "slellipses.h"
#include "slsplines.h"
```

```
// VertexList constructor

VertexList::VertexList() : BaseList() {
  // initialize operator identification numbers
  currentId = 1;
}


// Set current node in the list to the node containing v

void VertexList::SetCur(Vertex *v) {
  for (First();!AtEnd();Next())
    if (v == GetCur()->GetVertex()) return;
}


// Append the new node os to the tail of the list and
// assign the new node a unique id number;

void VertexList::Append(VertexNode *os) {
  BaseList::Append(os);
  if (os->GetVertex()->GetId())
    currentId = currentId > os->GetVertex()->GetId() ?
currentId : os->GetVertex()->GetId() + 1;
  else
    os->GetVertex()->SetId(currentId++);
}

// Set current node of list to be the one whose shape selection contains
// the given point

Vertex *VertexList::FindOp(Coord x, Coord y)  {
  Vertex *rval = 0;
  PointObj *p = new PointObj(x,y);
  for (First(); !AtEnd(); Next())
    if (GetCur()->GetVertex()->GetShape()->Contains(p)) {
      rval = GetCur()->GetVertex();
      break;
    }
  delete p;
  return rval;
}


// set the current node to the vertex whose id is vid

Vertex *VertexList::FindOp(int vid) {
  for (First(); !AtEnd(); Next())
    if (GetCur()->GetVertex()->GetId() == vid)
      return GetCur()->GetVertex();
  return 0;
}


// replace the operator component oldSel with newSel

void VertexList::Replace(Selection *oldSel, Selection *newSel) {
  Vertex *v = (Vertex *) oldSel->GetOwner();
  switch (oldSel->GetClassId()) {
    case TERMINATOR :
    case OPERATOR : v->SetShape(newSel);
      break;
    case LABEL_OP : v->SetLabel((TextSelection *)newSel);
      break;
    case MET_OP : ((Operator *)v)->SetMET((TextSelection *)newSel);
      break;
  }
}


// remove the indicated selection from an operator component

void VertexList::Remove(Selection *s) {
  Vertex *v = (Vertex *) s->GetOwner();
  switch (s->GetClassId()) {
```

```
    case TERMINATOR :
    case OPERATOR : SetCur(v);
      DeleteCur();
      delete v;
      break;
    case LABEL_OP :
      if (v) {
        v->SetLabel(nil);
        delete s;
  }
      break;
    case MET_OP :
      if (v) {
        ((Operator *)v)->SetMET(nil);
        delete s;
  }
      break;
   }
}

// set the current node to the one containing e

void EdgeList::SetCur(Edge *e) {
  for (First();!AtEnd();Next())
    if (e == GetCur()->GetEdge()) return;
}

// replace the Edge component oldSel with newSel

void EdgeList::Replace(Selection *oldSel, Selection *newSel) {
  Edge *e = (Edge *) oldSel->GetOwner();
  switch (oldSel->GetClassId()) {
    case SELFLOOP :
    case DATAFLOW_SPLINE :
      e->SetShape(newSel);
      break;
    case LABEL_SL :
    case LABEL_DF :
      e->SetLabel((TextSelection *)newSel);
      break;
    case LAT_DF :
      e->SetLatency((TextSelection *)newSel);
      break;
   }
}

// remove the Edge component s from the appropriate edge in the list

void EdgeList::Remove(Selection *s) {
  Edge *e = (Edge *) s->GetOwner();
  switch (s->GetClassId()) {
    case SELFLOOP :
    case DATAFLOW_SPLINE : SetCur(e);
      DeleteCur();
      delete e;
      break;
    case LABEL_SL :
    case LABEL_DF :
      if (e) {
        e->SetLabel(nil);
        delete s;
  }
      break;
    case LAT_DF :
      if (e) {
        e->SetLatency(nil);
        delete s;
  }
      break;
   }
}
```

```
// for all streams in the edgelist, output an appropriate
// type declaration

void EdgeList::WriteStreamTypes(FILE *fptr, char *hdr) {
  int count = 0;
  for (First();!AtEnd();Next()) {
    if (GetCur()->GetEdge()->IsState())
      continue;
    if (count++)
      fprintf(fptr,",\n");
    else
      fprintf(fptr,"%s",hdr);
    GetCur()->GetEdge()->WriteType(fptr);
   }
  if (count)
    fprintf(fptr,"\n");
}


// for all states in the list output an appropriate type declaration

void EdgeList::WriteStateTypes(FILE *fptr, char *hdr) {
  int count = 0;
  for (First();!AtEnd();Next()) {
    if (GetCur()->GetEdge()->IsStream())
      continue;
    if (count++)
      fprintf(fptr,",\n");
    else
      fprintf(fptr,"%s",hdr);
    GetCur()->GetEdge()->WriteType(fptr);
   }
  if (count)
    fprintf(fptr,"\n");
}
```

rubbands.h

```
#ifndef rubbands_h
#define rubbands_h

#include <InterViews/rubcurve.h>
#include <InterViews/rubline.h>
#include <InterViews/rubrect.h>

// An IStretchingRect uses its first few Track calls to decide which
// side it will let the user drag.

class IStretchingRect : public StretchingRect {
public:

   IStretchingRect(Painter*, Canvas*, Coord, Coord, Coord, Coord,
   Coord = 0, Coord = 0);

   void Track(Coord, Coord);
   void DefineSide(Coord, Coord);
   Alignment CurrentSide(boolean landscape);

protected:

   boolean firsttime;// stores true until after first call of Track
   boolean undefinedside;// stores true until side has been determined
   Coord cx, cy;// stores original center of rectangle
   Coord origx, origy;// stores point passed by first call of Track

};

// A RubberMultiLine lets the user drag one of its vertices.

class RubberMultiLine : public RubberVertex {
public:
   RubberMultiLine(
      Painter*, Canvas*, Coord px[], Coord py[], int n, int pt,
Coord offx = 0, Coord offy = 0
   );
```

```
    void Draw();
};

// A RubberPolygon lets the user drag one of its vertices.

class RubberPolygon : public RubberVertex {
public:
    RubberPolygon(
        Painter*, Canvas*, Coord px[], Coord py[], int n, int pt,
Coord offx = 0, Coord offy = 0
    );
    void Draw();
};

#endif
```

rubbands.c

```
#include "rubbands.h"
#include <InterViews/painter.h>

// abs returns the integer's magnitude.

inline int abs (int a) {
    return a > 0 ? a : -a;
}

// IStretchingRect starts with no defined side yet.

IStretchingRect::IStretchingRect (Painter* p, Canvas* c, Coord x0, Coord y0,
Coord x1, Coord y1, Coord offx, Coord offy)
: (p, c, x0, y0, x1, y1, RightSide, offx, offy) {
    firsttime = true;
    undefinedside = true;
    cx = (fixedx + movingx) / 2;
    cy = (fixedy + movingy) / 2;
}

// Track stores the point on the first call, decides which side
// becomes the stretching side on the next call or two, and draws the
// rectangle on all following calls.

void IStretchingRect::Track (Coord x, Coord y) {
    if (firsttime) {
firsttime = false;
origx = x;
origy = y;
    } else if (undefinedside) {
DefineSide(x, y);
    } else {
StretchingRect::Track(x, y);
    }
}

// DefineSide picks the side to stretch after the motion of the
// tracking point becomes sufficiently unambiguous.

static const Coord THRESHOLD = 2;

void IStretchingRect::DefineSide (Coord x, Coord y) {
    Coord dx = abs(x - origx);
    Coord dy = abs(y - origy);
    Coord xydiff = abs(dx - dy);
    if (xydiff >= THRESHOLD) {
undefinedside = false;
if (dx > dy) {
    if (x > cx) {
side = RightSide;
    } else {
side = LeftSide;
    }
} else {
    if (y > cy) {
side = TopSide;
    } else {
side = BottomSide;
    }
```

```
}
    }
}

// CurrentSide returns the side that the user is dragging.

Alignment IStretchingRect::CurrentSide (boolean landscape) {
    Alignment s = Left;
    switch (side) {
    case LeftSide:
s = landscape ? Bottom : Left;
break;
    case RightSide:
s = landscape ? Top : Right;
break;
    case TopSide:
s = landscape ? Right : Top;
break;
    case BottomSide:
s = landscape ? Left : Bottom;
break;
    }
    return s;
}

// RubberMultiLine passes its arguments to RubberVertex.

RubberMultiLine::RubberMultiLine (
    Painter* p, Canvas* c, Coord px[], Coord py[], int n, int pt,
    Coord offx, Coord offy
) : (p, c, px, py, n, pt, offx, offy) {
    /* nothing else to do */
}

// Draw draws only the one or two line segments attached to the rubber
// vertex.

void RubberMultiLine::Draw () {
    if (x == nil || y == nil) {
        return;
    }
    if (!drawn) {
int before = rubberPt - 1;
if (before >= 0) {
    Coord x0 = x[before];
    Coord y0 = y[before];
    output->Line(canvas, x0+offx, y0+offy, trackx+offx, tracky+offy);
}
int after = rubberPt + 1;
if (after <= count - 1) {
    Coord x1 = x[after];
    Coord y1 = y[after];
    output->Line(canvas, trackx+offx, tracky+offy, x1+offx, y1+offy);
}
drawn = true;
    }
}

// RubberPolygon passes its arguments to RubberVertex.

RubberPolygon::RubberPolygon (
    Painter* p, Canvas* c, Coord px[], Coord py[], int n, int pt,
    Coord offx, Coord offy
) : (p, c, px, py, n, pt, offx, offy) {
    /* nothing else to do */
}

// Draw draws only the two line segments attached to the rubber
// vertex.

void RubberPolygon::Draw () {
    if (x == nil || y == nil) {
        return;
    }
    if (!drawn) {
int before = (rubberPt > 0) ? rubberPt - 1 : count - 1;
```

```
Coord x0 = x[before];
Coord y0 = y[before];
output->Line(canvas, x0+offx, y0+offy, trackx+offx, tracky+offy);
if (count > 2) {
   int after = (rubberPt < count - 1) ? rubberPt + 1 : 0;
   Coord x1 = x[after];
   Coord y1 = y[after];
   output->Line(canvas, trackx+offx, tracky+offy, x1+offx, y1+offy);
}
drawn = true;
   }
}
```

selecter.h

```
/* Description:  definition of class which gives user ability to select a
 *           prototype name.
 *
 * Changes made by:  Mehdi Rowshanaee
 * Last change made:  October 1994

 */

#ifndef selecter_h
#define selecter_h

#include <InterViews/strchooser.h>
#include <InterViews/frame.h>
#include <InterViews/strbrowser.h>

/*
 * A Selecter allows the user to select the prototype he wants to use.
 */

class Selecter : public StringChooser {
public:
   Selecter(Interactor*, const char*, Alignment);
   char* Select();
void Insert(char**);
void SetErrorTitle(const char*);
   void Clear() {browser()->Clear();}

protected:
   void Init(const char*);
   Interactor* Interior();
   boolean Popup(Event&, boolean = true);

   Interactor* underlying;      // parent interactor that we'll overlay

private:
    Interactor* AddScroller(Interactor*);
    StringBrowser* browser() { return (StringBrowser*) _browser; }

    MarginFrame* title;
 MarginFrame* error_title;
 Alignment align;
};

#endif
```

selecter.c

```
#include "selecter.h"
#include <InterViews/button.h>
#include <InterViews/event.h>
#include <InterViews/frame.h>
#include <InterViews/message.h>
#include <InterViews/streditor.h>
#include <InterViews/world.h>
#include <InterViews/glue.h>
#include <InterViews/box.h>
#include <InterViews/border.h>
#include <InterViews/adjuster.h>
#include <InterViews/scroller.h>
#include <InterViews/sensor.h>
#include <InterViews/Std/string.h>
#define __string_h
```

```
#include "istring.h"
#include <stdio.h>
#include <sys/types.h>
#include <sys/dir.h>
/* #include <string.h> */

int compare_successful(char*, const char*);
char* define_extension(const char*);

void find_prototype_names(char** prototype_array, char* dir, const char* func)
 {
  DIR* pdir = opendir(dir);
  boolean successful = pdir != NULL;
  struct direct* d;
  char* name;
  int no_of_prototypes = 0;

  if (successful)
   {
    for (d = readdir(pdir); d != NULL; d = readdir(pdir))
     {
      int compare_result = compare_successful(d->d_name, func);
  if (compare_result)
        {
name = new char [compare_result + 1];
          strncpy(name, d->d_name, compare_result);
          name[compare_result] = '\0';
prototype_array[no_of_prototypes] = new char[strlen(name)+1];
strcpy(prototype_array[no_of_prototypes++],name);
        }
      }
    }
  prototype_array[no_of_prototypes] = nil;
  closedir(pdir);
  }

int compare_successful(char* filename, const char* func)
 {
  char* extension;
  extension = ".graph";

  int filename_len = strlen(filename);
  int ext_len = strlen(extension);

  if (filename_len < ext_len)
    return 0;
  else
   {
    int limit = filename_len - ext_len;
    for (int i = 0; i <= limit; ++i)
     {
      if (strncmp(filename + i, extension, ext_len) == 0)
        return i;
     }
    return 0;
   }
 }

Selecter::Selecter (Interactor* u, const char* t, Alignment a) :
 StringChooser(new ButtonState, 10, 24, "", a)
 {
  align = a;
  underlying = u;
  Init(t);
  Scene::Insert(Interior());
 }

char* Selecter::Select ()
 {
  char* name=nil;
  Event e;
  if (Popup(e))
    name = (char *)Choice();

  return name;
 }
```

```
void Selecter::Insert(char** name_array)
 {
  for (int i = 0; name_array[i] != nil; ++i)
   browser()->Append(name_array[i]);
 }

void Selecter::Init(const char* t)
 {
  if (*t == '\0')
    title = new MarginFrame(new VGlue(0,0));
  else
    title = new MarginFrame(new Message(t));
  error_title = new MarginFrame(new VGlue(0,0));
 }

Interactor* Selecter::Interior ()
 {
  const int space = round(.5 * cm);
VBox* errorblock = new VBox( new HBox(error_title, new HGlue) );

  VBox* titleblock = new VBox( new HBox(title, new HGlue) );

  return new Frame(new MarginFrame
            ( new VBox( errorblock, titleblock,
              new VGlue(space,0),new Frame(AddScroller(browser())),
              new VBox(
              new VGlue(space,0),
              new Frame(
              new MarginFrame(_sedit,2))),
              new VGlue(space,0),
              new HBox(
              new VGlue(space,0),
              new HGlue,
              new PushButton("Cancel", state, '\007'),
              new HGlue(space,0),
              new PushButton("Select", state, '\r')
               )
             ), space, space/2, 0
          ), 2
       );
 }

boolean Selecter::Popup (Event&, boolean)
 {
  World* world = underlying->GetWorld();
  Coord x, y;
  underlying->Align(align, 0, 0, x, y);
  underlying->GetRelative(x, y, world);
  world->InsertTransient(this, underlying, x, y, align);
  boolean accepted = Accept();
  world->Remove(this);
  return accepted;
 }

Interactor* Selecter::AddScroller(Interactor* i)
 {
  return new HBox(
       new MarginFrame(i,2),
       new VBorder,
       new VBox(
       new UpMover(i,1),
       new HBorder,
       new VScroller(i),
       new HBorder,
       new DownMover(i,1)
      ) );
 }

void ChangeMsg(const char* name, MarginFrame* frame)
 {
  Interactor* msg;
  if (*name == '\0')
msg = new VGlue(0,0);
  else
msg = new Message(name);
```

```
  frame->Insert(msg);
  frame->Change(msg);
  }

void Selecter::SetErrorTitle(const char* name)
 {
  ChangeMsg(name, error_title);
  }
```

selection.h

```
#define KERNEL

#ifndef selection_h
#define selection_h

#include <InterViews/Graphic/picture.h>

// Declare imported types.

class GraphComponent;
class Rubberband;
class RubberVertex;
class State;
class istream;
class ostream;

// PostScript file format changes.

static const int ORIGINALVERSION = 1;  // original format
static const int FGCOLORVERSION = 2;  // added foreground color
static const int NONREDUNDANTVERSION = 3; // eliminated unnecessary text
 // pattern and duplication of
 // font name, transformation matrix,
 // poly points, and text data
static const int FGANDBGCOLORVERSION = 4; // added background color and
 // RGB values for overriding names;
 // used graylevel to eliminate
 // redundant patternfill data
static const int GRIDSPACINGVERSION = 5;  // added grid spacing
static const int NONROTATEDVERSION = 6;  // replaced rotation of drawing with
 // rotation of view for landscape
static const int TEXTOFFSETVERSION = 7;  // changed text positions on screen
 // and improved accuracy of
 // text positions on printout

// Other constants.

static const int ARROWHEIGHT = 8;     // how long arrows are in points
static const int ARROWWIDTH = 4;      // how wide arrows are in points
static const int BUFSIZE = 256;       // size of buffer for reading data
static const int HDSIZE = 5;      // how wide handles are in points

// A Selection can draw handles around itself and create a reshaped
// copy of itself.

extern const char* startdata;      // signals place to read valid data

class Selection : public Picture {
public:

   Selection(ClassId, Graphic* = nil);
Selection(Graphic* = nil);
   ~Selection();

   Graphic* Copy();
   boolean HasChildren();

   void GetPaddedBox(BoxObj&);
   void DrawHandles(Painter*, Canvas*);
   void EraseHandles(Painter*, Canvas*);
   void RedrawHandles(Painter*, Canvas*);
   void RedrawUnclippedHandles(Painter*, Canvas*);
   void ResetHandles();

   virtual boolean ShapedBy(Coord, Coord, float);
```

```
    virtual Rubberband* CreateShape(Coord, Coord);
    virtual Selection* GetReshapedCopy();

    virtual void WriteData(ostream&);

      void SetClassId(ClassId);
      ClassId GetClassId();
   boolean IsEdgeComponent();
   boolean IsVertexComponent();
   void SetOwner(GraphComponent *o) {owner = o;}
   GraphComponent* GetOwner() {return owner;}

 protected:

    void Skip(istream&);
    void ReadVersion(istream&);
    void ReadGridSpacing(istream&, State*);
    void ReadGS(istream&, State*);
    void ReadPictGS(istream&, State*);
    void ReadTextGS(istream&, State*);
    void ReadBrush(istream&, State*);
    void ReadFgColor(istream&, State*);
    void ReadBgColor(istream&, State*);
    void ReadFont(istream&, State*);
    void ReadPattern(istream&, State*);
    void ReadTransformer(istream&);
    float CalcGrayLevel(int);

    void WriteVersion(ostream&);
    void WriteGridSpacing(ostream&, State*);
    void WriteGS(ostream&);
    void WritePictGS(ostream&);
    void WriteTextGS(ostream&);
    void WriteBrush(ostream&);
    void WriteFgColor(ostream&);
    void WriteBgColor(ostream&);
    void WriteFont(ostream&);
    void WritePattern(ostream&);
    void WriteTransformer(ostream&);

    virtual void CreateHandles();
    void DeleteHandles();

    Rubberband* handles;// contains handles outlining Selection

    static char buf[BUFSIZE];// contains storage for reading data
    static int versionnumber;// stores version of drawing read from file
   ClassId cid;            // stores type of selection
   GraphComponent *owner;
};

// An NPtSelection knows how to create handles, read or write its
// points, and draw arrowheads so many subclasses can reuse the same
// code.

class NPtSelection : public Selection {
public:

   NPtSelection(ClassId, Graphic*);
   NPtSelection(Graphic*);
   virtual int GetOriginal(const Coord*&, const Coord*&);
   boolean ShapedBy(Coord, Coord, float);
   Rubberband* CreateShape(Coord, Coord);
   Selection* GetReshapedCopy();
ClassId GetClassId();

protected:

   void ReadPoints(istream&, const Coord*&, const Coord*&, int&);
   void WriteData(ostream&);

   virtual RubberVertex* CreateRubberVertex(Coord*, Coord*, int, int);
   virtual Selection* CreateReshapedCopy(Coord*, Coord*, int);
   void CreateHandles();
   void TotalTransform(Coord*, Coord*, int);
   void InvTotalTransform(Coord*, Coord*, int);
```

```
    int ClosestPoint(Coord*, Coord*, int, Coord, Coord);

    boolean LeftAcont(Coord, Coord, Coord, Coord, PointObj&, Graphic*);
    boolean RightAcont(Coord, Coord, Coord, Coord, PointObj&, Graphic*);
    boolean LeftAints(Coord, Coord, Coord, Coord, BoxObj&, Graphic*);
    boolean RightAints(Coord, Coord, Coord, Coord, BoxObj&, Graphic*);
    void drawLeftA(Coord, Coord, Coord, Coord, Canvas*, Graphic*);
    void drawRightA(Coord, Coord, Coord, Coord, Canvas*, Graphic*);
    float MergeArrowHeadTol(float, Graphic*);
    boolean ArrowHeadcont(Coord, Coord, Coord, Coord, PointObj&, Graphic*);
    boolean ArrowHeadints(Coord, Coord, Coord, Coord, BoxObj&, Graphic*);
    void drawArrowHead(Coord, Coord, Coord, Coord, Canvas*, Graphic*);
    void SetCTM(Coord, Coord, Coord, Coord, Graphic*, boolean);
    void RestoreCTM(Graphic*);
    float Slope(float, float);

    const char* myname;// tells which NPtSelection subclass this is
    RubberVertex* rubbervertex;// tells us how the user wants to reshape us

};

#endif
```

selection.c

```
#define _POSIX_SOURCE
#define KERNEL
#include "dfdclasses.h"
#include "ipaint.h"
#include "istring.h"
#include "listifont.h"
#include "mapipaint.h"
#include "selection.h"
#include "state.h"
#include <InterViews/Std/math.h>
#include <InterViews/rubcurve.h>
#include <InterViews/transformer.h>
#include <InterViews/Graphic/util.h>
#include <stream.h>
#define KERNEL

// #include <InterViews/Std/stdio.h>


/*
 * Define the start of data token.
 */

const char* startdata = "%I";

/*
 * Selection starts off with no handles and class id set.
 */

Selection::Selection (ClassId classid, Graphic* gs) : (gs)
{
  handles = nil;
  cid = classid;
  owner = 0;
}

/*
 * Selection starts off with no handles and unknown class id.
 */

Selection::Selection (Graphic* gs) : (gs)
{
  handles = nil;
  cid = NONE;
  owner = 0;
}

/*
 * Free storage allocated for the handles if any.
 */
```

```
Selection::~Selection ()
{
  DeleteHandles();
}

/*
 * Copy returns a copy of the Selection.
 */

Graphic* Selection::Copy ()
{
  return new Selection(this);
}

/*
 * HasChildren returns false so Idraw won't ungroup this Picture.
 */

boolean Selection::HasChildren ()
{
  return false;
}

/*
 * GetPaddedBox returns the Selection's smallest box with enough
 * padding added to include its handles.
 */

void Selection::GetPaddedBox (BoxObj& box)
{
  Picture::GetBox(box);
  const int HDPAD = HDSIZE/2 + 1; /* how much to add to GetBox's size */
  box.left -= HDPAD;
  box.bottom -= HDPAD;
  box.right += HDPAD;
  box.top += HDPAD;
}

/*
 * DrawHandles tells the handles to draw themselves unless they've
 * already drawn themselves.
 */

void Selection::DrawHandles (Painter* rasterxor, Canvas* canvas)
{
  if (handles == nil)
    {
CreateHandles();
    }
  handles->SetPainter(rasterxor);
  handles->SetCanvas(canvas);
  handles->Draw();
}

/*
 * EraseHandles tells the handles to erase themselves unless they've
 * already erased themselves.
 */

void Selection::EraseHandles (Painter* rasterxor, Canvas* canvas)
{
  if (handles != nil)
    {
handles->SetPainter(rasterxor);
handles->SetCanvas(canvas);
handles->Erase();
    }
}

/*
 * RedrawHandles knows for sure that no unerased handles remain on the
 * screen, so it resets the handles to outline the Selection's
 * possibly different shape and location before drawing the handles.
 */
```

```
void Selection::RedrawHandles (Painter* rasterxor, Canvas* canvas)
{
  DeleteHandles();
  DrawHandles(rasterxor, canvas);
}

/*
 * RedrawUnclippedHandles knows that some unerased handles probably
 * remain on the screen so it can't reset the handles, but it can tell
 * the handles to draw themselves whether or not they've already drawn
 * themselves because the painter will clip the already drawn handles.
 */

void Selection::RedrawUnclippedHandles (Painter* rasterxor, Canvas* canvas)
{
  if (handles == nil)
    {
CreateHandles();
    }
  handles->SetPainter(rasterxor);
  handles->SetCanvas(canvas);
  handles->Redraw();
}

/*
 * ResetHandles deletes the handles since the Selection may have moved
 * out from under them.  Redrawing the handles will recreate them.
 */

void Selection::ResetHandles ()
{
  DeleteHandles();
}

/*
 * ShapedBy returns false since the Selection does not contain any
 * points which both determine its shape and fall within the given
 * distance of the given point.
 */

boolean Selection::ShapedBy (Coord, Coord, float)
{
  return false;
}

/*
 * CreateShape creates and returns a Rubberband representing the
 * Selection's shape for the user to reshape.
 */

Rubberband* Selection::CreateShape (Coord, Coord)
{
  return nil;
}

/*
 * GetReshapedCopy creates and returns a copy of the Selection
 * incorporating the change made to its shape.
 */

Selection* Selection::GetReshapedCopy ()
{
  return nil;
}

/*
 * Skip skips over tokens in the input stream until it reads a start
 * of data token or reaches eof.
 */

void Selection::Skip (istream& from)
{
  while (from >> buf && strcmp(buf, startdata) != 0)
    {
/* skip Postscript code */
    }
```

```
}

/*
 * ReadVersion reads the drawing's version number.  Knowing the
 * drawing's version number allows us to invoke backward compatibility
 * code if necessary or detect an incompatibility ahead of time.
 */

void Selection::ReadVersion (istream& from)
 {
   Skip(from);
   from >> buf;
   if (strcmp(buf, "Idraw") == 0)
     {
from >> versionnumber;
     }
   else
     {
versionnumber = ORIGINALVERSION;
     }
   if (versionnumber > TEXTOFFSETVERSION)
     {
fprintf(stderr, "warning: drawing version %d ", versionnumber);
fprintf(stderr, "newer than idraw version %d\n", TEXTOFFSETVERSION);
     }
}

/*
 * ReadGridSpacing reads the grid spacing used by the drawing and
 * stores the new grid spacing value.  It must correct the default
 * grid spacing it gives to old drawings for an implementation botch
 * in InterViews 2.4 that calculated point's value incorrectly using
 * 72.07/inch instead of inch/72.27 (it was a botch in TWO ways).
 */

void Selection::ReadGridSpacing (istream& from, State* state)
{
   double g = state->GetGridSpacing();
   if (versionnumber < GRIDSPACINGVERSION)
     {
const int oldspacing = 8;
const double oldpoints = 72.07/inches;
g = oldpoints * round(oldspacing * oldpoints);
     }
   else
     {
from >> buf;
if (strcmp(buf, "Grid") == 0)
       {
   from >> g;
 }
     }
   state->SetGridSpacing(g);
}

/*
 * ReadGS reads data to initialize the graphic state for Selections
 * which don't contain any text.
 */

void Selection::ReadGS (istream& from, State* state)
 {
   ReadBrush(from, state);
   if (versionnumber >= FGANDBGCOLORVERSION)
     {
ReadFgColor(from, state);
ReadBgColor(from, state);
SetFont(nil);
     }
   else if (versionnumber >= FGCOLORVERSION)
     {
ReadFgColor(from, state);
IColor* bg = state->GetMapIBgColor()->GetInitial();
SetColors(GetFgColor(), bg);
SetFont(nil);
     }
```

```
    else
      {
IColor* fg = state->GetMapIFgColor()->GetInitial();
IColor* bg = state->GetMapIBgColor()->GetInitial();
SetColors(fg, bg);
ReadFont(from, state);
      }
    ReadPattern(from, state);
    ReadTransformer(from);
}

/*
 * ReadPictGS reads data to initialize the graphic state for
 * PictSelections which may contain some text.
 */

void Selection::ReadPictGS (istream& from, State* state)
 {
    ReadBrush(from, state);
    if (versionnumber >= FGANDBGCOLORVERSION)
      {
ReadFgColor(from, state);
ReadBgColor(from, state);
      }
    else if (versionnumber >= FGCOLORVERSION)
      {
ReadFgColor(from, state);
SetColors(GetFgColor(), nil);
      }
    else
      {
SetColors(nil, nil);
      }
    ReadFont(from, state);
    ReadPattern(from, state);
    ReadTransformer(from);
}

/*
 * ReadTextGS reads data to initialize the graphic state for
 * TextSelections which don't need a brush or pattern.
 */

void Selection::ReadTextGS (istream& from, State* state)
 {
    if (versionnumber >= FGCOLORVERSION)
      {
SetBrush(nil);
ReadFgColor(from, state);
SetColors(GetFgColor(), nil);
      }
    else
      {
ReadBrush(from, state);
SetColors(state->GetMapIFgColor()->GetInitial(), nil);
      }
    ReadFont(from, state);
    if (versionnumber < NONREDUNDANTVERSION)
      {
ReadPattern(from, state);
IPattern* pattern = (IPattern*) GetPattern();
float graylevel = pattern->GetGrayLevel();
const char* c = "Black";
int r = 0, g = 0, b = 0;
if (graylevel != 0 && graylevel != -1)
      {
    if (graylevel == 1)
          {
c = "White";
r = g = b = 65535;
      }
          else
            {
c = "Gray";
r = g = b = 49152;
      }
```

```
  }
SetColors(state->GetMapIFgColor()->FindOrAppend(c, r, g, b), nil);
    }
   else
    {
SetPattern(nil);
    }
   ReadTransformer(from);
   if (versionnumber < TEXTOFFSETVERSION)
    {
IFont* f = (IFont*) GetFont();
float x0, y0, x1, y1;
transform(0.0, 0.0, x0, y0);
transform(0.0, float(f->GetLineHt() - f->Height() - 1), x1, y1);
float dx = x1 - x0;
float dy = y1 - y0;
Translate(dx, dy);
    }
  }

/*
 * ReadBrush reads data to set the Selection's brush.
 */

void Selection::ReadBrush (istream& from, State* state)
 {
   Skip(from);
   from >> buf;
   if (buf[0] == 'b')
    {
char lookahead = 'u';
boolean undefined = false;
boolean none = false;
int p = 0;
int w = 0;
int l = false;
int r = false;

from >> lookahead;
from.putback(lookahead);
switch (lookahead)
      {
   case 'u':
     undefined = true;
     break;
   case 'n':
     none = true;
     break;
   default:
     from >> p >> w >> l >> r;
     break;
   }

if (undefined || !from.good())
      {
   SetBrush(nil);
  }
     else
      {
   MapIBrush* mb = state->GetMapIBrush();
   IBrush* brush = mb->FindOrAppend(none, p, w, l, r);
   SetBrush(brush);
  }
    }
  }

/*
 * ReadFgColor reads data to set the Selection's foreground color.
 */

void Selection::ReadFgColor (istream& from, State* state)
 {
   Skip(from);
   from >> buf;
   if (buf[0] == 'c' &&
(buf[1] == 'f' || versionnumber < FGANDBGCOLORVERSION) )
```

```
    {
char lookahead = 'u';
boolean undefined = false;
char name[100];
float fr = 0, fg = 0, fb = 0;

from >> lookahead;
from.putback(lookahead);
if (lookahead == 'u')
      {
  undefined = true;
  }
     else
      {
  from >> name;
  if (versionnumber >= FGANDBGCOLORVERSION)
        {
from >> fr >> fg >> fb;
    }
  }

if (undefined || !from.good())
      {
  SetColors(nil, GetBgColor());
 }
     else
      {
  int r = round(fr * 0xffff);
  int g = round(fg * 0xffff);
  int b = round(fb * 0xffff);
  MapIColor* mfg = state->GetMapIFgColor();
  IColor* fgcolor = mfg->FindOrAppend(name, r, g, b);
  SetColors(fgcolor, GetBgColor());
 }
    }
 }

/*
 * ReadBgColor reads data to set the Selection's background color.
 */

void Selection::ReadBgColor (istream& from, State* state)
 {
  Skip(from);
  from >> buf;
  if (buf[0] == 'c' && buf[1] == 'b')
   {
char lookahead = 'u';
boolean undefined = false;
char name[100];
float fr = 0, fg = 0, fb = 0;

from >> lookahead;
from.putback(lookahead);
if (lookahead == 'u')
      {
  undefined = true;
 }
     else
      {
  from >> name >> fr >> fg >> fb;
 }

if (undefined || !from.good())
      {
  SetColors(GetFgColor(), nil);
 }
     else
      {
  int r = round(fr * 0xffff);
  int g = round(fg * 0xffff);
  int b = round(fb * 0xffff);
  MapIColor* mbg = state->GetMapIBgColor();
  IColor* bgcolor = mbg->FindOrAppend(name, r, g, b);
  SetColors(GetFgColor(), bgcolor);
 }
```

```
    }
  }

 /*
  * ReadFont reads data to set the Selection's font.
  */

 void Selection::ReadFont (istream& from, State* state)
  {
   Skip(from);
   from >> buf;
   if (buf[0] == 'f')
    {
 char lookahead = 'u';
 boolean undefined = false;
 char name[100];
 char printfont[100];
 char printsize[100];

 from >> lookahead;
 from.putback(lookahead);
 if (lookahead == 'u')
       {
   undefined = true;
  }
      else
       {
   from >> name;
   from >> printfont;
   from >> printsize;
  }

 if (undefined || !from.good())
       {
   SetFont(nil);
  }
      else
       {
   MapIFont* mf = state->GetMapIFont();
   char* pf = (versionnumber >= NONREDUNDANTVERSION) ?
&printfont[1] : printfont;
   IFont* font = mf->FindOrAppend(name, pf, printsize);
   SetFont(font);
  }
    }
  }

 /*
  * ReadPattern reads data to set the Selection's pattern.
  */

 void Selection::ReadPattern (istream& from, State* state)
  {
   Skip(from);
   from >> buf;
   if (buf[0] == 'p')
    {
 char lookahead = 'u';
 boolean undefined = false;
 boolean none = false;
 float graylevel = 0;
 int data[patternHeight];
 int size = 0;

 from >> lookahead;
 switch (lookahead)
       {
   case 'u':
    undefined = true;
    break;
   case 'n':
    none = true;
    break;
   case '<':
    graylevel = -1;
    break;
```

```
    default:
    from.putback(lookahead);
    break;
  }

if (!undefined && !none && graylevel != -1)
        {
    if (versionnumber >= FGANDBGCOLORVERSION)
          {
from >> graylevel;
      }
         else
          {
from >> data[0];
graylevel = CalcGrayLevel(data[0]);
           }
  }
      else if (graylevel == -1)
        {
    for (int i = 0; from >> buf && i < patternHeight; i++)
          {
if ((buf[0] == '>') || (sscanf(buf, "%x", &data[i]) != 1))
            {
     break;
  }
      }
    if (buf[0] == '>')
          {
size = i;
      }
         else
          {
undefined = true;
      }
    }

if (undefined || !from.good())
        {
    SetPattern(nil);
  }
      else
        {
    MapIPattern* mp = state->GetMapIPattern();
    IPattern* pattern = mp->FindOrAppend(none, graylevel, data, size);
    SetPattern(pattern);
  }
     }
  }

/*
 * ReadTransformer reads data to set the Selection's transformation
 * matrix.
 */

void Selection::ReadTransformer (istream& from)
  {
    Skip(from);
    from >> buf;
    if (buf[0] == 't')
      {
char uorbracket = 'u';
boolean undefined = false;
float a00, a01, a10, a11, a20, a21;

from >> uorbracket;
if (uorbracket == 'u')
        {
    undefined = true;
  }
      else
        {
    if (versionnumber < NONREDUNDANTVERSION)
          {
from.putback(uorbracket);
      }
    from >> a00 >> a01 >> a10 >> a11 >> a20 >> a21;
```

```
  }

if (from.good() && !undefined)
      {
  SetTransformer(new Transformer(a00, a01, a10, a11, a20, a21));
 }
    }
 }

/*
 * CalcGrayLevel calculates a 4x4 bitmap's gray level on the printer.
 * Since the gray level ranges from 0 = solid to 1 = clear,
 * CalcGrayLevel counts the number of 0 bits in the bitmap and divides
 * the sum by the total number of bits in the bitmap.
 */

float Selection::CalcGrayLevel (int seed)
 {
  const int numbits = 16;
  int numzeros = 0;
  for (int i = 0; i < numbits; i++)
   {
numzeros += !((seed >> i) & 0x1);
    }
  return float(numzeros) / numbits;
 }

/*
 * WriteData writes everything needed to draw or reconstruct the
 * Selection.
 */

void Selection::WriteData (ostream& to)
{
  /* define it in your subclass */
}

/*
 * WriteVersion writes the drawing's version number.  Storing a
 * version number with the drawing makes backward compatibility easier
 * to support when the drawing format changes in the future.
 */

void Selection::WriteVersion (ostream& to)
{
  to << startdata << " Idraw " << TEXTOFFSETVERSION << " ";
}

/*
 * WriteGridSpacing writes the drawing's grid spacing.  Storing the
 * grid spacing with the drawing ensures graphics will remain aligned
 * to the grid they were aligned to before.
 */

void Selection::WriteGridSpacing (ostream& to, State* state)
{
  to << "Grid " << state->GetGridSpacing() << " ";
}

/*
 * WriteGS writes the graphic state for Selections that don't contain
 * any text.
 */

void Selection::WriteGS (ostream& to)
{
  WriteBrush(to);
  WriteFgColor(to);
  WriteBgColor(to);
  WritePattern(to);
  WriteTransformer(to);
}

/*
 * WritePictGS writes the graphic state for PictSelections which may
 * contain some text.
```

```
*/

void Selection::WritePictGS (ostream& to)
{
   WriteBrush(to);
   WriteFgColor(to);
   WriteBgColor(to);
   WriteFont(to);
   WritePattern(to);
   WriteTransformer(to);
}

/*
 * WriteTextGS writes the graphic state for TextSelections which
 * don't need a brush or pattern but do need a font.
 */

void Selection::WriteTextGS (ostream& to)
{
   WriteFgColor(to);
   WriteFont(to);
   WriteTransformer(to);
}

/*
 * WriteBrush writes the Selection's brush.
 */

void Selection::WriteBrush (ostream& to)
 {
   IBrush* brush = (IBrush*) GetBrush();
   if (brush == nil)
     {
to << startdata << " b u\n";
     }
   else if (brush->None())
     {
to << "none SetB " << startdata << " b n\n";
     }
   else
     {
int p = brush->GetLinePattern();
to << startdata << " b " << p << "\n";
int w = brush->Width();
boolean l = brush->LeftArrow();
boolean r = brush->RightArrow();
to << w << " " << l << " " << r << " ";
const int* dashpat = brush->GetDashPattern();
int dashpatsize = brush->GetDashPatternSize();
int dashoffset = brush->GetDashOffset();
if (dashpatsize <= 0)
       {
   to << "[] " << dashoffset << " ";
  }
      else
        {
   to << "[";
   for (int i = 0; i < dashpatsize - 1; i++)
          {
to << dashpat[i] << " ";
     }
   to << dashpat[i] << "] " << dashoffset << " ";
  }
to << "SetB\n";
     }
  }

/*
 * WriteFgColor writes the Selection's foreground color.
 */

void Selection::WriteFgColor (ostream& to)
  {
   IColor* fgcolor = (IColor*) GetFgColor();
   if (fgcolor == nil)
     {
```

```
to << startdata << " cfg u\n";
    }
   else
    {
const char* name = fgcolor->GetName();
to << startdata << " cfg " << name << "\n";
if (strcmp(name, "white") == 0 || strcmp(name, "White") == 0)
      {
  to << "1 1 1 SetCFg\n";
  }
     else
      {
  int r, g, b;
  fgcolor->Intensities(r, g, b);
  float fr = float(r) / 0xffff;
  float fg = float(g) / 0xffff;
  float fb = float(b) / 0xffff;
  to << fr << " " << fg << " " << fb << " SetCFg\n";
 }
   }
 }

/*
 * WriteBgColor writes the Selection's background color.
 */

void Selection::WriteBgColor (ostream& to)
 {
  IColor* bgcolor = (IColor*) GetBgColor();
  if (bgcolor == nil)
   {
to << startdata << " cbg u\n";
    }
   else
    {
const char* name = bgcolor->GetName();
to << startdata << " cbg " << name << "\n";
if (strcmp(name, "white") == 0 || strcmp(name, "White") == 0)
      {
  to << "1 1 1 SetCBg\n";
 }
     else
      {
  int r, g, b;
  bgcolor->Intensities(r, g, b);
  float fr = float(r) / 0xffff;
  float fg = float(g) / 0xffff;
  float fb = float(b) / 0xffff;
  to << fr << " " << fg << " " << fb << " SetCBg\n";
 }
   }
 }

/*
 * WriteFont writes the Selection's font.
 */

void Selection::WriteFont (ostream& to)
 {
  IFont* font = (IFont*) GetFont();
  if (font == nil)
   {
to << startdata << " f u\n";
    }
   else
    {
const char* name = font->GetName();
const char* pf = font->GetPrintFont();
const char* ps = font->GetPrintSize();
to << startdata << " f " << name << "\n";
to << "/" << pf << " " << ps << " SetF\n";
    }
 }

/*
 * WritePattern writes the Selection's pattern.
```

```
*/

void Selection::WritePattern (ostream& to)
 {
  IPattern* pattern = (IPattern*) GetPattern();
  if (pattern == nil)
    {
to << startdata << " p u\n";
    }
  else if (pattern->None())
    {
to << "none SetP " << startdata << " p n\n";
    }
  else if (pattern->GetSize() > 0)
    {
const int* data = pattern->GetData();
int size = pattern->GetSize();
to << startdata << " p\n";
to << "< ";
if (size <= 8)
      {
  for (int i = 0; i < 8; i++)
        {
sprintf(buf, "%02x", data[i] & 0xff);
to << buf << " ";
    }
 }
     else
      {
  for (int i = 0; i < patternHeight; i++)
        {
sprintf(buf, "%0*x", patternWidth/4, data[i]);
if (i != patternHeight - 2)
            {
  to << buf << " ";
 }
           else
            {
  to << buf << "\n  ";
 }
    }
 }
to << "> -1 SetP\n";
    }
  else
    {
float graylevel = pattern->GetGrayLevel();
to << startdata << " p\n";
to << graylevel << " SetP\n";
    }
 }

/*
 * WriteTransformer writes the Selection's transformation matrix.
 */

void Selection::WriteTransformer (ostream& to)
 {
  Transformer* t = GetTransformer();
  if (t == nil || *t == *identity)
    {
to << startdata << " t u\n";
    }
  else
    {
float mat[6];
t->GetEntries(mat[0], mat[1], mat[2], mat[3], mat[4], mat[5]);
to << startdata << " t\n[ ";
for (int i = 0; i < 6; i++)
      {
  to << (fabs(mat[i]) < 0.0001 ? 0.0 : mat[i]) << " ";
 }
to << "] concat\n";
    }
 }
```

```
/*
 * CreateHandles creates handles outlining the Selection's shape.
 */

void Selection::CreateHandles ()
 {
  const int N = 8;
  const int RUBPT = 0;
  Coord l, b, r, t, hx, hy, x[N], y[N];

  Picture::GetBox(l, b, r, t);
  hx = (r + l)/2;
  hy = (t + b)/2;
  x[0] = l;y[0] = b;
  x[1] = hx;y[1] = b;
  x[2] = r;y[2] = b;
  x[3] = r;y[3] = hy;
  x[4] = r;y[4] = t;
  x[5] = hx;y[5] = t;
  x[6] = l;y[6] = t;
  x[7] = l;y[7] = hy;

  handles = new RubberHandles(nil, nil, x, y, N, RUBPT, HDSIZE);
 }

/*
 * DeleteHandles zeroes the handles to record it has none now.
 */

void Selection::DeleteHandles ()
 {
  if (handles != nil)
    {
delete handles;
handles = nil;
    }
 }

/*
 * SetClassId describes the type of selection it is storing
 */

void Selection::SetClassId(ClassId classid)
{
cid = classid;
}

/*
 * GetClassId returns the type of selection it is storing
 */

ClassId Selection::GetClassId()
{
return cid;
}

boolean Selection::IsEdgeComponent() {
  return (cid == SELFLOOP) || (cid == DATAFLOW_SPLINE) ||
       (cid == LAT_DF) || (cid == LABEL_DF) ||
       (cid == LABEL_SL);
}

boolean Selection::IsVertexComponent() {
  return (cid == OPERATOR) || (cid == LABEL_OP) ||
       (cid == MET_OP);
}

/*
 * NPtSelection passes its argument to Selection.
 */

NPtSelection::NPtSelection (ClassId classid, Graphic* gs) : (classid, gs)
{
  myname = "YouForgotToDefineMyName";
  rubbervertex = nil;
}
```

```
NPtSelection::NPtSelection(Graphic* gs) : (gs)
{
myname = "YouForgotToDefineMyName";
rubbervertex = nil;
}

/*
 * GetOriginal returns the points that were passed to the
 * the NPtSelection subclass's constructor.
 */

int NPtSelection::GetOriginal (const Coord*&, const Coord*&)
{
   return 0;
}

/*
 * GetClassId returns the type of selection it is storing
 */

ClassId NPtSelection::GetClassId()
{
return Selection::GetClassId();
}

/*
 * ShapedBy returns true if any of the NPtSelection's points falls
 * within the given distance of the given point.
 */

boolean NPtSelection::ShapedBy (Coord px, Coord py, float maxdist)
{
   const Coord* ux;
   const Coord* uy;
   int n = GetOriginal(ux, uy);
   Coord* x = new Coord[n];
   Coord* y = new Coord[n];
   CopyArray(ux, uy, n, x, y);
   TotalTransform(x, y, n);
   int closestpt = ClosestPoint(x, y, n, px, py);
   boolean shapedby = Distance(x[closestpt], y[closestpt], px, py) <= maxdist;
   delete x;
   delete y;
   return shapedby;
}

/*
 * CreateShape creates, stores, and returns a rubberband representing
 * the NPtSelection's shape for the user to reshape.
 */

Rubberband* NPtSelection::CreateShape (Coord px, Coord py)
{
   const Coord* ux;
   const Coord* uy;
   int n = GetOriginal(ux, uy);
   Coord* x = new Coord[n];
   Coord* y = new Coord[n];
   CopyArray(ux, uy, n, x, y);
   TotalTransform(x, y, n);
   int rubpt = ClosestPoint(x, y, n, px, py);
   rubbervertex = CreateRubberVertex(x, y, n, rubpt);
   delete x;
   delete y;
   return rubbervertex;
}

/*
 * GetReshapedCopy creates and returns a copy of the NPtSelection
 * incorporating the change made to its shape.
 */

Selection* NPtSelection::GetReshapedCopy ()
{
   Coord* x;
```

```
    Coord* y;
    int n;
    int rubpt;
    rubbervertex->GetCurrent(x, y, n, rubpt);
    delete rubbervertex;

    InvTotalTransform(x, y, n);
    Selection* reshaped = CreateReshapedCopy(x, y, n);
    delete x;
    delete y;
    return reshaped;
}

/*
 * ReadPoints reads a set of points as efficiently as possible by
 * using dynamic static buffers instead of mallocing on every call.
 */

void NPtSelection::ReadPoints (istream& from, const Coord*& x, const Coord*& y,
int& n)
 {
   const int INITIALSIZE = 15;
   static int sizepoints = 0;
   static Coord* xcoords = nil;
   static Coord* ycoords = nil;

   Skip(from);
   from >> n;
   if (n > sizepoints)
    {
delete xcoords;
delete ycoords;
sizepoints = max(n, INITIALSIZE);
xcoords = new Coord[sizepoints];
ycoords = new Coord[sizepoints];
    }

   for (int i = 0; i < n; i++)
    {
if (versionnumber < NONREDUNDANTVERSION)
      {
   Skip(from);
  }
from >> xcoords[i] >> ycoords[i];
    }

   x = xcoords;
   y = ycoords;
 }

/*
 * WriteData writes the NPtSelection's data and Postscript code to
 * draw it.
 */

void NPtSelection::WriteData (ostream& to)
 {
   const Coord* x;
   const Coord* y;
   int n = GetOriginal(x, y);
   to << "Begin " << startdata << " " << myname << "\n";
   WriteGS(to);
   to << startdata << " " << n << "\n";
   for (int i = 0; i < n; i++)
    {
to << x[i] << " " << y[i] << "\n";
    }
   to << n << " " << myname << "\n";
   to << "End\n\n";
 }

/*
 * CreateRubberVertex creates and returns the right kind of
 * RubberVertex to represent the NPtSelection's shape.
 */
```

```
RubberVertex* NPtSelection::CreateRubberVertex (Coord*, Coord*, int, int)
 {
  /* implement it in your subclass */
  return nil;
}

/*
 * CreateReshapedCopy creates and returns a reshaped copy of itself
 * using the passed points and its graphic state.
 */

Selection* NPtSelection::CreateReshapedCopy (Coord*, Coord*, int)
{
  /* implement it in your subclass */
  return nil;
}

/*
 * CreateHandles creates handles highlighting the NPtSelection's
 * points.
 */

void NPtSelection::CreateHandles ()
 {
  const int RUBPT = 0;
  const Coord* ux;
  const Coord* uy;
  int n = GetOriginal(ux, uy);
  Coord* x = new Coord[n];
  Coord* y = new Coord[n];
  CopyArray(ux, uy, n, x, y);
  TotalTransform(x, y, n);
  handles = new RubberHandles(nil, nil, x, y, n, RUBPT, HDSIZE);
  delete x;
  delete y;
 }

/*
 * TotalTransform transforms the given points from the Selection's
 * coordinate system to the screen's coordinate system.
 */

void NPtSelection::TotalTransform (Coord* x, Coord* y, int n)
 {
  Transformer total;

  TotalTransformation(total);
  for (int i = 0; i < n; i++)
   {
total.Transform(x[i], y[i]);
   }
 }

/*
 * InvTotalTransform transforms the given points from the screen's
 * coordinate system to the NPtSelection's coordinate system.
 */

void NPtSelection::InvTotalTransform (Coord* x, Coord* y, int n)
 {
  Transformer total;

  TotalTransformation(total);
  for (int i = 0; i < n; i++)
   {
total.InvTransform(x[i], y[i]);
   }
 }

/*
 * ClosestPoint returns the index within the arrays of the closest
 * point to the given coordinates.
 */

int NPtSelection::ClosestPoint (Coord* x, Coord* y, int n, Coord px,
Coord py)
```

```
  {
   int closestpt = 0;
   float mindist = Distance(x[0], y[0], px, py);
   for (int i = 1; i < n; i++)
    {
float dist = Distance(x[i], y[i], px, py);
if (dist < mindist)
      {
  mindist = dist;
  closestpt = i;
 }
    }
  return closestpt;
 }

/*
 * LeftAcont checks whether the left arrowhead contains the point.
 */

boolean NPtSelection::LeftAcont (Coord x0, Coord y0, Coord x1, Coord y1,
PointObj& po, Graphic* gs)
 {
  IBrush* brush = (IBrush*) gs->GetBrush();
  if (brush->LeftArrow())
   {
return ArrowHeadcont(x0, y0, x1, y1, po, gs);
   }
  return false;
 }

/*
 * RightAcont checks whether the right arrowhead contains the point.
 */

boolean NPtSelection::RightAcont (Coord x0, Coord y0, Coord x1, Coord y1,
PointObj& po, Graphic* gs)
 {
  IBrush* brush = (IBrush*) gs->GetBrush();
  if (brush->RightArrow())
   {
return ArrowHeadcont(x0, y0, x1, y1, po, gs);
   }
  return false;
 }

/*
 * LeftAints checks whether the left arrowhead intersects the area.
 */

boolean NPtSelection::LeftAints (Coord x0, Coord y0, Coord x1, Coord y1,
BoxObj& userb, Graphic* gs)
 {
  IBrush* brush = (IBrush*) gs->GetBrush();
  if (brush->LeftArrow())
   {
return ArrowHeadints(x0, y0, x1, y1, userb, gs);
   }
  return false;
 }

/*
 * RightAints checks whether the right arrowhead intersects the area.
 */

boolean NPtSelection::RightAints (Coord x0, Coord y0, Coord x1, Coord y1,
BoxObj& userb, Graphic* gs)
 {
  IBrush* brush = (IBrush*) gs->GetBrush();
  if (brush->RightArrow())
   {
return ArrowHeadints(x0, y0, x1, y1, userb, gs);
   }
  return false;
 }

/*
```

```
 * drawLeftA draws the left arrowhead if it should be drawn.
 */

void NPtSelection::drawLeftA (Coord x0, Coord y0, Coord x1, Coord y1,
Canvas* c, Graphic* gs)
 {
  IBrush* brush = (IBrush*) gs->GetBrush();
  if (brush->LeftArrow())
    {
drawArrowHead(x0, y0, x1, y1, c, gs);
    }
 }

/*
 * drawRightA draws the right arrowhead if it should be drawn.
 */

void NPtSelection::drawRightA (Coord x0, Coord y0, Coord x1, Coord y1,
Canvas* c, Graphic* gs)
 {
  IBrush* brush = (IBrush*) gs->GetBrush();
  if (brush->RightArrow())
    {
drawArrowHead(x0, y0, x1, y1, c, gs);
    }
 }

/*
 * Define an arrow head with its tip at the origin and its tail on the
 * negative x axis so we can rotate the tail about the origin to the
 * right angle and translate the tip to the right point.
 */

static const int ARROWN = 3;
static Coord arrowx[ARROWN] = {0, -ARROWHEIGHT, -ARROWHEIGHT};
static Coord arrowy[ARROWN] = {0, ARROWWIDTH/2, -ARROWWIDTH/2};
static Coord arrowconvx[ARROWN + 1];
static Coord arrowconvy[ARROWN + 1];

/*
 * MergeArrowHeadTol returns a tolerance to use around the graphic's
 * extent that includes the arrowhead's size.
 */

float NPtSelection::MergeArrowHeadTol (float tol, Graphic* gs)
 {
  IBrush* brush = (IBrush*) gs->GetBrush();
  if (brush->LeftArrow() || brush->RightArrow())
    {
float magnif = 1;
Transformer* view = getRoot()->GetTransformer();
if (view != nil && !view->Rotated())
      {
       /* rot breaks magnif calc */
  float fpx0, fpy0, fpx1, fpy1;
  view->Transform(0.0, 0.0, fpx0, fpy0);
  view->Transform(1.0, 1.0, fpx1, fpy1);
  magnif = fpy1 - fpy0;
 }
float arrowtol = 0.5 * ARROWWIDTH * points * magnif;
tol = max(arrowtol, tol);
    }
  return tol;
 }

/*
 * ArrowHeadcont returns true if the arrowhead contains the point.
 * The arrowhead may be filled or unfilled depending on the pattern.
 */

boolean NPtSelection::ArrowHeadcont (Coord x0, Coord y0, Coord x1, Coord y1,
PointObj& po, Graphic* gs)
 {
  boolean contains = false;
  IPattern* pattern = (IPattern*) gs->GetPattern();
```

```
    SetCTM(x0, y0, x1, y1, gs, !pattern->None());
    PointObj pt(&po);
    invTransform(pt.x, pt.y, gs);

    if (pattern->None())
      {
  MultiLineObj ml(arrowx, arrowy, ARROWN);
  LineObj l(arrowx[ARROWN-1], arrowy[ARROWN-1], arrowx[0], arrowy[0]);
  contains = ml.Contains(pt) || l.Contains(pt);
      }
    else
      {
  FillPolygonObj fp(arrowx, arrowy, ARROWN);
  contains = fp.Contains(pt);
      }

    RestoreCTM(gs);
    return contains;
  }

/*
 * ArrowHeadints returns true if the arrowhead intersects the area.
 * The arrowhead may be filled or unfilled depending on the pattern.
 */

boolean NPtSelection::ArrowHeadints (Coord x0, Coord y0, Coord x1, Coord y1,
BoxObj& userb, Graphic* gs)
  {
    boolean intersects = false;
    IPattern* pattern = (IPattern*) gs->GetPattern();

    SetCTM(x0, y0, x1, y1, gs, !pattern->None());
    transformList(arrowx, arrowy, ARROWN, arrowconvx, arrowconvy, gs);

    if (pattern->None())
      {
  arrowconvx[ARROWN] = arrowconvx[0];
  arrowconvy[ARROWN] = arrowconvy[0];
  MultiLineObj ml(arrowconvx, arrowconvy, ARROWN + 1);
  intersects = ml.Intersects(userb);
      }
    else
      {
  FillPolygonObj fp(arrowconvx, arrowconvy, ARROWN);
  intersects = fp.Intersects(userb);
      }

    RestoreCTM(gs);
    return intersects;
  }

/*
 * drawArrowHead draws the arrowhead.
 */

void NPtSelection::drawArrowHead (Coord x0, Coord y0, Coord x1, Coord y1,
Canvas* c, Graphic* gs)
  {
    IPattern* pattern = (IPattern*) gs->GetPattern();
    if (!pattern->None())
      {
  SetCTM(x0, y0, x1, y1, gs, true);
  update(gs);
  pFillPolygon(c, arrowx, arrowy, ARROWN);
  RestoreCTM(gs);
      }

    IBrush* brush = (IBrush*) gs->GetBrush();
    if (!brush->None())
      {
  SetCTM(x0, y0, x1, y1, gs, false);
  update(gs);
  pPolygon(c, arrowx, arrowy, ARROWN);
  RestoreCTM(gs);
      }
  }
```

```
/*
 * SetCTM stores gs's former transformation matrix and overwrites it
 * with a new one defined to scale the arrowhead to its proper size
 * and align it with the line. The matrix includes the graphic's
 * topmost parent's scaling but no other parents' scaling so the
 * arrowhead will change size when we zoom the view but stay the same
 * size when we scale the line. New argument pattemfill special-
 * cases filling arrowhead to include the outermost edge of the
 * outline drawn by the brush so dashed brushes won't expose white
 * space where there's no pattern fill.
 */

static Transformer* origCTM;
static Transformer* arrowCTM;

void NPtSelection::SetCTM (Coord x0, Coord y0, Coord x1, Coord y1,
Graphic* gs, boolean pattemfill)
 {
  if (arrowCTM == nil)
    {
arrowCTM = new Transformer;
arrowCTM->Reference();
    }
  *arrowCTM = *identity;

  if (pattemfill)
    {
IBrush* brush = (IBrush*) gs->GetBrush();
float bw = brush->Width();
float padtip = sqrt(ARROWHEIGHT*ARROWHEIGHT +
  0.25*ARROWWIDTH*ARROWWIDTH) * bw / ARROWWIDTH;
float padtail = bw / 2;
float patternscale = (ARROWHEIGHT + padtip + padtail) / ARROWHEIGHT;
arrowCTM->Scale(patternscale, patternscale);
arrowCTM->Translate(padtip, 0.);
    }

  arrowCTM->Scale(point, point);
  Transformer* view = getRoot()->GetTransformer();
  if (view != nil && !view->Rotated())
    {
     /* rot breaks magnif calc */
float fpx0, fpy0, fpx1, fpy1;
view->Transform(0.0, 0.0, fpx0, fpy0);
view->Transform(1.0, 1.0, fpx1, fpy1);
float arrowxmag = fpx1 - fpx0;
float arrowymag = fpy1 - fpy0;
if (arrowxmag == arrowymag)
      {
       /* won't scale arrows in BrushView */
  arrowCTM->Scale(arrowxmag, arrowymag);
 }
    }

  float tipx, tipy, tailx, taily;
  transform(float(x0), float(y0), tipx, tipy, gs);
  transform(float(x1), float(y1), tailx, taily, gs);
  float angle = Slope(tipx - tailx, tipy - taily);
  arrowCTM->Rotate(angle);
  arrowCTM->Translate(tipx, tipy);

  origCTM = gs->GetTransformer();
  if (origCTM != nil)
    {
origCTM->Reference();
    }
  gs->SetTransformer(arrowCTM);
 }

/*
 * RestoreCTM restores gs's original transformation matrix.
 */

void NPtSelection::RestoreCTM (Graphic* gs)
 {
```

```
    gs->SetTransformer(origCTM);
    Unref(origCTM);
  }

/*
 * sign returns 1 if the number's nonnegative, -1 if it's negative.
 */

inline float sign (float num)
  {
    return (num >= 0.) ? 1. : -1.;
  }

/*
 * Slope returns the number of degrees in the given slope.
 */

float NPtSelection::Slope (float dx, float dy)
  {
    float angle = 0.;
    if (dx == 0.)
      {
        angle = sign(dy) * 90.;
      }
    else
      {
angle = degrees(atan(dy/dx));
if (dx < 0.)
        {
    angle += sign(dy) * 180.;
  }
      }
    return angle;
  }
```

slellipses.h

```
#ifndef slellipses_h
#define slellipses_h

#include "selection.h"

class Operator;

// A EllipseSelection draws a ellipse with an outline and a filled
// interior.

class EllipseSelection : public Selection {
public:

    EllipseSelection(Coord, Coord, Coord, Coord, Graphic* = nil);
    EllipseSelection(istream&, State*);

    Graphic* Copy();
    void GetOriginal(Coord&, Coord&, int&, int&);

protected:

    void WriteData(ostream&);

};

// A CircleSelection draws a circle with an outline and a filled
// interior.

class CircleSelection : public Selection {
public:

    CircleSelection(Coord, Coord, int, Graphic* = nil);
    CircleSelection(istream&, State*);

    Graphic* Copy();
    void GetOriginal(Coord&, Coord&, int&);

protected:
```

```
    void WriteData(ostream&);

};

#endif
```

slellipses.c

```
#include "dfdclasses.h"
#include "iellipses.h"
#include "slellipses.h"
#include <InterViews/Std/stream.h>

// EllipseSelection creates the ellipse's filled interior and outline.

EllipseSelection::EllipseSelection (Coord x0, Coord y0, int rx, int ry,
Graphic* gs) : (OPERATOR, gs) {
    Append(new IFillEllipse(x0, y0, rx, ry));
    Append(new Ellipse(x0, y0, rx, ry));
}

// EllipseSelection reads data to initialize its graphic state and
// create its filled interior and outline.

EllipseSelection::EllipseSelection (istream& from, State* state) :
    Selection((Graphic *)nil) {
    ReadGS(from, state);
    Skip(from);
    Coord x0, y0;
    int rx, ry;
    from >> x0 >> y0 >> rx >> ry;
    Append(new IFillEllipse(x0, y0, rx, ry));
    Append(new Ellipse(x0, y0, rx, ry));
}

// Copy returns a copy of the EllipseSelection.

Graphic* EllipseSelection::Copy () {
    Coord x0, y0;
    int rx, ry;
    GetOriginal(x0, y0, rx, ry);
    return new EllipseSelection(x0, y0, rx, ry, this);
}

// GetOriginal returns the center point and the x and y radii lengths
// that were passed to the EllipseSelection's constructor.

void EllipseSelection::GetOriginal (Coord& x0, Coord& y0, int& rx, int& ry) {
    ((Ellipse*) Last())->GetOriginal(x0, y0, rx, ry);
}

// WriteData writes the EllipseSelection's data and Postscript code to
// draw it.

void EllipseSelection::WriteData (ostream& to) {
    Coord x0, y0;
    int rx, ry;
    GetOriginal(x0, y0, rx, ry);
    to << "Begin " << startdata << " Elli\n";
    WriteGS(to);
    to << startdata << "\n";
    to << x0 << " " << y0 << " " << rx << " " << ry << " Elli\n";
    to << "End\n\n";
}

// CircleSelection creates the circle's filled interior and outline.

CircleSelection::CircleSelection (Coord x0, Coord y0, int r, Graphic* gs)
: (gs) {
    Append(new IFillCircle(x0, y0, r));
    Append(new Circle(x0, y0, r));
}

// CircleSelection reads data to initialize its graphic state and
// create its filled interior and outline.
```

```
CircleSelection::CircleSelection (istream& from, State* state) :
Selection ((Graphic*)nil) {
    ReadGS(from, state);
    Skip(from);
    Coord x0, y0;
    int r;
    from >> x0 >> y0 >> r;
    Append(new IFillCircle(x0, y0, r));
    Append(new Circle(x0, y0, r));
}

// Copy returns a copy of the CircleSelection.

Graphic* CircleSelection::Copy () {
    Coord x0, y0;
    int r;
    GetOriginal(x0, y0, r);
    return new CircleSelection(x0, y0, r, this);
}

// GetOriginal returns the center point and radius length that were
// passed to the CircleSelection's constructor.

void CircleSelection::GetOriginal (Coord& x0, Coord& y0, int& r) {
    ((Circle*) Last())->GetOriginal(x0, y0, r, r);
}

// WriteData writes the CircleSelection's data and Postscript code to
// draw it.

void CircleSelection::WriteData (ostream& to) {
    Coord x0, y0;
    int r;
    GetOriginal(x0, y0, r);
    to << "Begin " << startdata << " Circ\n";
    WriteGS(to);
    to << startdata << "\n";
    to << x0 << " " << y0 << " " << r << " Circ\n";
    to << "End\n\n";
}
```

sllines.h

```
#ifndef sllines_h
#define sllines_h

#include "selection.h"

// A LineSelection draws a line.

class LineSelection : public NPtSelection {
public:

    LineSelection(Coord, Coord, Coord, Coord, Graphic* = nil);
    LineSelection(istream&, State*);

    Graphic* Copy();
    void GetOriginal2(Coord&, Coord&, Coord&, Coord&);
    int GetOriginal(const Coord*&, const Coord*&);
Selection* CreateReshapedCopy(Coord*, Coord*, int);

protected:

    void WriteData(ostream&);
    RubberVertex* CreateRubberVertex(Coord*, Coord*, int, int);

    void uncacheChildren();
    void getExtent(float&, float&, float&, float&, float&, Graphic*);
    boolean contains(PointObj&, Graphic*);
    boolean intersects(BoxObj&, Graphic*);
    void draw(Canvas*, Graphic*);
    void drawClipped(Canvas*, Coord, Coord, Coord, Coord, Graphic*);

    Graphic* line;// draws the line
```

```
};

// A MultiLineSelection draws a set of connected lines with a filled
// interior.

class MultiLineSelection : public NPtSelection {
public:

   MultiLineSelection(Coord*, Coord*, int, Graphic* = nil);
   MultiLineSelection(istream&, State*);

   Graphic* Copy();
   int GetOriginal(const Coord*&, const Coord*&);

protected:

   void Init(Coord*, Coord*, int);
   RubberVertex* CreateRubberVertex(Coord*, Coord*, int, int);
   Selection* CreateReshapedCopy(Coord*, Coord*, int);

   void uncacheChildren();
   void getExtent(float&, float&, float&, float&, float&, Graphic*);
   boolean contains(PointObj&, Graphic*);
   boolean intersects(BoxObj&, Graphic*);
   void draw(Canvas*, Graphic*);
   void drawClipped(Canvas*, Coord, Coord, Coord, Coord, Graphic*);

   Graphic* ifillmultiline;// fills a set of connected lines
   Graphic* multiline;// draws a set of connected lines
   Coord lx0, ly0, lx1, ly1;// stores endpoints of left arrowhead
   Coord rx0, ry0, rx1, ry1;// stores endpoints of right arrowhead

};

#endif
```

slpict.h

```
#ifndef slpict_h
#define slpict_h

#include "selection.h"

// Declare imported types.

class IFont;
class IFontList;

// A PictSelection contains other Selections.

class PictSelection : public Selection {
public:

   PictSelection(Graphic* = nil);
   PictSelection(FILE*, State*);

   Graphic* Copy();
   boolean HasChildren();
   void Propagate();

   boolean Valid();
   boolean WritePicture(FILE*, State*, boolean);

   Selection* GetCurrent();
   Selection* First();
   Selection* Last();
   Selection* Next();
   Selection* Prev();

   Selection* FirstSelectionContaining(PointObj&);
   Selection* LastSelectionContaining(PointObj&);
   int SelectionsContaining(PointObj&, Selection**&);

   Selection* FirstSelectionIntersecting(BoxObj&);
   Selection* LastSelectionIntersecting(BoxObj&);
```

```
    int SelectionsIntersecting(BoxObj&, Selection**&);

    Selection* FirstSelectionWithin(BoxObj&);
    Selection* LastSelectionWithin(BoxObj&);
    int SelectionsWithin(BoxObj&, Selection**&);

  int FindIndex(Selection*);
  Selection* GetSelection(int);
  protected:

    PictSelection(istream&, State*);
    void ReadChildren(istream&, State*);

    void WritePicture(ostream&, State*, boolean);
    void WriteComments(ostream&);
    void WritePrologue(ostream&);
    void WriteDrawing(ostream&);
    void WriteData(ostream&);
    void WriteTrailer(ostream&);
    void ScaleToPostscriptCoords();
    void ScaleToScreenCoords();
    void CollectFonts(IFontList*);
    void Merge(IFont*, IFontList*);

    boolean valid;// true if creation of PictSelection succeeded

  };

  // Define inline access functions to get members' values.

  inline boolean PictSelection::Valid () {
    return valid;
  }

  // Cast these functions to return Selections instead of Graphics.

  inline Selection* PictSelection::GetCurrent() {
    return (Selection*) Picture::GetCurrent();
  }

  inline Selection* PictSelection::First() {
    return (Selection*) Picture::First();
  }

  inline Selection* PictSelection::Last() {
    return (Selection*) Picture::Last();
  }

  inline Selection* PictSelection::Next() {
    return (Selection*) Picture::Next();
  }

  inline Selection* PictSelection::Prev() {
    return (Selection*) Picture::Prev();
  }

  inline Selection* PictSelection::FirstSelectionContaining(PointObj& p) {
    return (Selection*) Picture::FirstGraphicContaining(p);
  }

  inline Selection* PictSelection::LastSelectionContaining(PointObj& p) {
    return (Selection*) Picture::LastGraphicContaining(p);
  }

  inline int PictSelection::SelectionsContaining(PointObj& p, Selection**& ss) {
    Graphic** gg = nil;
    int num = Picture::GraphicsContaining(p, gg);
    ss = (Selection**) gg;
    return num;
  }

  inline Selection* PictSelection::FirstSelectionIntersecting(BoxObj& b) {
    return (Selection*) Picture::FirstGraphicIntersecting(b);
  }

  inline Selection* PictSelection::LastSelectionIntersecting(BoxObj& b) {
```

```
    return (Selection*) Picture::LastGraphicIntersecting(b);
}

inline int PictSelection::SelectionsIntersecting(BoxObj& b, Selection**& ss) {
    Graphic** gg = nil;
    int num = Picture::GraphicsIntersecting(b, gg);
    ss = (Selection**) gg;
    return num;
}

inline Selection* PictSelection::FirstSelectionWithin(BoxObj& b) {
    return (Selection*) Picture::FirstGraphicWithin(b);
}

inline Selection* PictSelection::LastSelectionWithin(BoxObj& b) {
    return (Selection*) Picture::LastGraphicWithin(b);
}

inline int PictSelection::SelectionsWithin(BoxObj& b, Selection**& ss) {
    Graphic** gg = nil;
    int num = Picture::GraphicsWithin(b, gg);
    ss = (Selection**) gg;
    return num;
}

#endif
```

slpict.c

```
#include "ipaint.h"
#include "istring.h"
#include "listifont.h"
#include "slellipses.h"
#include "sllines.h"
#include "slpict.h"
#include "slpolygons.h"
#include "slsplines.h"
#include "sltext.h"
#include <InterViews/transformer.h>
#include <stream.h>

// PictSelection initializes its graphic state.

PictSelection::PictSelection (Graphic* gs) : (gs) {
    valid = true;
}

// PictSelection knows it's the outermost PictSelection because it was
// called with a FILE* pointer, so it must read a version number, skip
// over its name, read its graphic state and children, and scale
// itself back to screen coordinates when it's finished.

PictSelection::PictSelection (FILE* stream, State* state) : Selection((Graphic*) nil) {
    int fd = fileno(stream);
    istream from(fd);
    ReadVersion(from);
    ReadGridSpacing(from, state);
    if (versionnumber < NONREDUNDANTVERSION) {
Skip(from);
    }
    ReadPictGS(from, state);
    ReadChildren(from, state);
    ScaleToScreenCoords();
    Transformer* t = GetTransformer();
    if (versionnumber < NONROTATEDVERSION && t != nil && t->Rotated90()) {
*t = identity;
Translate(0.0, -8.5*inches);
Rotate(90.0, 0.0, 0.0);
Picture::Propagate();
    }
    valid = from.good();
}

// Copy returns a copy of the PictSelection.

Graphic* PictSelection::Copy () {
```

```
    Selection* copy = new PictSelection(this);
    for (First(); !AtEnd(); Next()) {
copy->Append(GetCurrent()->Copy());
    }
    return copy;
}

// HasChildren returns true so Idraw can ungroup this Picture.

boolean PictSelection::HasChildren () {
    return Picture::HasChildren();
}

// Propagate must preserve the PictSelection's transformation matrix
// if it has any.

void PictSelection::Propagate () {
    Transformer* original = GetTransformer();
    if (original != nil) {
original->Reference();
Picture::Propagate();
SetTransformer(original);
Unref(original);
    } else {
Picture::Propagate();
    }
}

// WritePicture writes the picture and returns true if the write
// succeeded or false if some IO error occurred. It omits the
// Postscript prologue and trailer if called with verbose false to
// speed up cutting and pasting pictures between drawings.

boolean PictSelection::WritePicture (
    FILE* stream, State* state, boolean verbose
) {
#if !defined(__GNUG__)
    // incompatible with g++, but works around a cfront bug
    filebuf fb(stream);
    ostream to(&fb);
#else
    int fd = fileno(stream);
    ostream to(fd);
#endif
    WritePicture(to, state, verbose);
    return to.good();
}

// PictSelection knows it's not the outermost PictSelection so it only
// reads data to initialize its graphic state and create its children
// Selections.

PictSelection::PictSelection (istream& from, State* state) : Selection((Graphic*)nil) {
    ReadPictGS(from, state);
    ReadChildren(from, state);
    valid = from.good();
}

// ReadChildren loops determining which kind of Selection follows and
// creating it until it reads "end" which means all of the children
// have been created.

void PictSelection::ReadChildren (istream& from, State* state) {
    while (from.good()) {
Skip(from);
Selection* child = nil;
from >> buf;
if (strcmp(buf, "BSpl") == 0) {
    child = new BSplineSelection(from, state);
} else if (strcmp(buf, "Circ") == 0) {
    child = newCircleSelection(from, state);
} else if (strcmp(buf, "CBSpl") == 0) {
    child = newClosedBSplineSelection(from, state);
} else if (strcmp(buf, "Elli") == 0) {
    child = newEllipseSelection(from, state);
} else if (strcmp(buf, "Line") == 0) {
```

```
    child = newLineSelection(from, state);
  } else if (strcmp(buf, "MLine") == 0) {
    child = newMultiLineSelection(from, state);
  } else if (strcmp(buf, "Pict") == 0) {
    child = newPictSelection(from, state);
  } else if (strcmp(buf, "Poly") == 0) {
    child = newPolygonSelection(from, state);
  } else if (strcmp(buf, "Rect") == 0) {
    child = newRectSelection(from, state);
  } else if (strcmp(buf, "Text") == 0) {
    child = newTextSelection(from, state);
  } else if (strcmp(buf, "eop") == 0) {
    break;
  } else {
    fprintf(stderr, "unknown Selection %s, skipping\n", buf);
    continue;
  }
  if (from.good()) {
    Append(child);
  } else {
    delete child;
  }
    }
}

// WritePicture writes the picture's data and Postscript code to print
// it wrapped in Postscript comments that minimally conform to version
// 1.0 of Adobe Systems's structuring conventions for Postscript.  The
// picture must remove itself from its parent if it has a parent to
// prevent the parent's transformation from affecting the picture's
// calculation of its bounding box.

void PictSelection::WritePicture (ostream& to, State* state, boolean verbose) {
    Picture* parent = (Picture*) Parent();
    if (parent != nil) {
  parent->SetCurrent(this);
  parent->Remove(this);
    }

    ScaleToPostscriptCoords();
    if (verbose) {
  WriteComments(to);
  WritePrologue(to);
  WriteVersion(to);
  WriteGridSpacing(to, state);
  WriteDrawing(to);
  WriteTrailer(to);
    } else {
  WriteVersion(to);
  WriteDrawing(to);
    }
    ScaleToScreenCoords();

    if (parent != nil) {
  parent->InsertBeforeCur(this);
    }
}

// WriteComments writes information about the picture such as the
// fonts used in it and the smallest bounding box enclosing it.

void PictSelection::WriteComments (ostream& to) {
    to << "%!PS-Adobe-2.0 EPSF-1.2\n";

    to << "%%DocumentFonts:";
    int linelen = strlen("%%DocumentFonts:");
    const int MAXLINELEN = 256;
    IFontList* fontlist = new IFontList;
    CollectFonts(fontlist);
    for (fontlist->First(); !fontlist->AtEnd(); fontlist->Next()) {
  IFont* font = fontlist->GetCur()->GetFont();
  if (linelen + strlen(font->GetPrintFont()) + 2 <= MAXLINELEN) {
    to << " ";
    ++linelen;
  } else {
    to << "\n%%+ ";
```

```
    linelen = strlen("%%+ ");
  }
  to << font->GetPrintFont();
  linelen += strlen(font->GetPrintFont());
    }
    to << "\n";
    delete fontlist;

    to << "%%Pages: 1\n";

    Coord l, b, r, t;
    GetBox(l, b, r, t);
    to << "%%BoundingBox: " << l << " " << b << " " << r << " " << t << "\n";

    to << "%%EndComments\n\n";
    to << "50 dict begin\n\n";
  }

  // WritePrologue writes definitions of Postscript procedures to draw
  // Selections.  You should not rename or delete the "exported"
  // capitalized procedures because old drawings rely on them, but you
  // can rename or delete the "internal" uncapitalized procedures.

  void PictSelection::WritePrologue (ostream& to) {
    to << "/arrowHeight " << ARROWHEIGHT << " def\n";
    to << "/arrowWidth " << ARROWWIDTH << " def\n";
    to << "/none null def\n";
    to << "/numGraphicParameters 17 def\n";
    to << "/stringLimit 65535 def\n\n";
    to << "/Begin {\n";
    to << "save\n";
    to << "numGraphicParameters dict begin\n";
    to << "} def\n\n";
    to << "/End {\n";
    to << "end\n";
    to << "restore\n";
    to << "} def\n\n";
    to << "/SetB {\n";
    to << "dup type /nulltype eq {\n";
    to << "pop\n";
    to << "false /brushRightArrow idef\n";
    to << "false /brushLeftArrow idef\n";
    to << "true /brushNone idef\n";
    to << "} {\n";
    to << "/brushDashOffset idef\n";
    to << "/brushDashArray idef\n";
    to << "0 ne /brushRightArrow idef\n";
    to << "0 ne /brushLeftArrow idef\n";
    to << "/brushWidth idef\n";
    to << "false /brushNone idef\n";
    to << "} ifelse\n";
    to << "} def\n\n";
    to << "/SetCFg {\n";
    to << "/fgblue idef\n";
    to << "/fggreen idef\n";
    to << "/fgred idef\n";
    to << "} def\n\n";
    to << "/SetCBg {\n";
    to << "/bgblue idef\n";
    to << "/bggreen idef\n";
    to << "/bgred idef\n";
    to << "} def\n\n";
    to << "/SetF {\n";
    to << "/printSize idef\n";
    to << "/printFont idef\n";
    to << "} def\n\n";
    to << "/SetP {\n";
    to << "dup type /nulltype eq {\n";
    to << "pop true /patternNone idef\n";
    to << "} {\n";
    to << "/patternGrayLevel idef\n";
    to << "patternGrayLevel -1 eq {\n";
    to << "/patternString idef\n";
    to << "} if\n";
    to << "false /patternNone idef\n";
    to << "} ifelse\n";
```

```
to << "} def\n\n";
to << "/BSpl {\n";
to << "0 begin\n";
to << "storexyn\n";
to << "newpath\n";
to << "n 1 gt {\n";
to << "0 0 0 0 0 0 1 1 true subspline\n";
to << "n 2 gt {\n";
to << "0 0 0 0 1 1 2 2 false subspline\n";
to << "1 1 n 3 sub {\n";
to << "/i exch def\n";
to << "i 1 sub dup i dup i 1 add dup i 2 add dup false subspline\n";
to << "} for\n";
to << "n 3 sub dup n 2 sub dup n 1 sub dup 2 copy false subspline\n";
to << "} if\n";
to << "n 2 sub dup n 1 sub dup 2 copy 2 copy false subspline\n";
to << "patternNone not brushLeftArrow not brushRightArrow not and and { ";
to << "ifill } if\n";
to << "brushNone not { istroke } if\n";
to << "0 0 1 1 leftarrow\n";
to << "n 2 sub dup n 1 sub dup rightarrow\n";
to << "} if\n";
to << "end\n";
to << "} dup 0 4 dict put def\n\n";
to << "/Circ {\n";
to << "newpath\n";
to << "0 360 arc\n";
to << "patternNone not { ifill } if\n";
to << "brushNone not { istroke } if\n";
to << "} def\n\n";
to << "/CBSpl {\n";
to << "0 begin\n";
to << "dup 2 gt {\n";
to << "storexyn\n";
to << "newpath\n";
to << "n 1 sub dup 0 0 1 1 2 2 true subspline\n";
to << "1 1 n 3 sub {\n";
to << "/i exch def\n";
to << "i 1 sub dup i dup i 1 add dup i 2 add dup false subspline\n";
to << "} for\n";
to << "n 3 sub dup n 2 sub dup n 1 sub dup 0 0 false subspline\n";
to << "n 2 sub dup n 1 sub dup 0 0 1 1 false subspline\n";
to << "patternNone not { ifill } if\n";
to << "brushNone not { istroke } if\n";
to << "} {\n";
to << "Poly\n";
to << "} ifelse\n";
to << "end\n";
to << "} dup 0 4 dict put def\n\n";
to << "/Elli {\n";
to << "0 begin\n";
to << "newpath\n";
to << "4 2 roll\n";
to << "translate\n";
to << "scale\n";
to << "0 0 1 0 360 arc\n";
to << "patternNone not { ifill } if\n";
to << "brushNone not { istroke } if\n";
to << "end\n";
to << "} dup 0 1 dict put def\n\n";
to << "/Line {\n";
to << "0 begin\n";
to << "2 storexyn\n";
to << "newpath\n";
to << "x 0 get y 0 get moveto\n";
to << "x 1 get y 1 get lineto\n";
to << "brushNone not { istroke } if\n";
to << "0 0 1 1 leftarrow\n";
to << "0 0 1 1 rightarrow\n";
to << "end\n";
to << "} dup 0 4 dict put def\n\n";
to << "/MLine {\n";
to << "0 begin\n";
to << "storexyn\n";
to << "newpath\n";
to << "n 1 gt {\n";
```

```
to << "x 0 get y 0 get moveto\n";
to << "1 1 n 1 sub {\n";
to << "/i exch def\n";
to << "x i get y i get lineto\n";
to << "} for\n";
to << "patternNone not brushLeftArrow not brushRightArrow not and and { ";
to << "ifill } if\n";
to << "brushNone not { istroke } if\n";
to << "0 0 1 1 leftarrow\n";
to << "n 2 sub dup n 1 sub dup rightarrow\n";
to << "} if\n";
to << "end\n";
to << "} dup 0 4 dict put def\n\n";
to << "/Poly {\n";
to << "3 1 roll\n";
to << "newpath\n";
to << "moveto\n";
to << "-1 add\n";
to << "{ lineto } repeat\n";
to << "closepath\n";
to << "patternNone not { ifill } if\n";
to << "brushNone not { istroke } if\n";
to << "} def\n\n";
to << "/Rect {\n";
to << "0 begin\n";
to << "/t exch def\n";
to << "/r exch def\n";
to << "/b exch def\n";
to << "/l exch def\n";
to << "newpath\n";
to << "l b moveto\n";
to << "l t lineto\n";
to << "r t lineto\n";
to << "r b lineto\n";
to << "closepath\n";
to << "patternNone not { ifill } if\n";
to << "brushNone not { istroke } if\n";
to << "end\n";
to << "} dup 0 4 dict put def\n\n";
to << "/Text {\n";
to << "ishow\n";
to << "} def\n\n";
to << "/idef {\n";
to << "dup where { pop pop pop } { exch def } ifelse\n";
to << "} def\n\n";
to << "/ifill {\n";
to << "0 begin\n";
to << "gsave\n";
to << "patternGrayLevel -1 ne {\n";
to << "fgred bgred fgred sub patternGrayLevel mul add\n";
to << "fggreen bggreen fggreen sub patternGrayLevel mul add\n";
to << "fgblue bgblue fgblue sub patternGrayLevel mul add setrgbcolor\n";
to << "eofill\n";
to << "} {\n";
to << "eoclip\n";
to << "originalCTM setmatrix\n";
to << "pathbbox /t exch def /r exch def /b exch def /l exch def\n";
to << "/w r l sub ceiling cvi def\n";
to << "/h t b sub ceiling cvi def\n";
to << "/imageByteWidth w 8 div ceiling cvi def\n";
to << "/imageHeight h def\n";
to << "bgred bggreen bgblue setrgbcolor\n";
to << "eofill\n";
to << "fgred fggreen fgblue setrgbcolor\n";
to << "w 0 gt h 0 gt and {\n";
to << "l b translate w h scale\n";
to << "w h true [w 0 0 h neg 0 h] { patternproc } imagemask\n";
to << "} if\n";
to << "} ifelse\n";
to << "grestore\n";
to << "end\n";
to << "} dup 0 8 dict put def\n\n";
to << "/istroke {\n";
to << "gsave\n";
to << "brushDashOffset -1 eq {\n";
to << "[] 0 setdash\n";
```

```
to << "1 setgray\n";
to << "} {\n";
to << "brushDashArray brushDashOffset setdash\n";
to << "fgred fggreen fgblue setrgbcolor\n";
to << "} ifelse\n";
to << "brushWidth setlinewidth\n";
to << "originalCTM setmatrix\n";
to << "stroke\n";
to << "grestore\n";
to << "} def\n\n";
to << "/ishow {\n";
to << "0 begin\n";
to << "gsave\n";
to << "fgred fggreen fgblue setrgbcolor\n";
to << "/fontDict printFont findfont printSize scalefont dup setfont def\n";
to << "/descender fontDict begin 0 [FontBBox] 1 get FontMatrix end\n";
to << "transform exch pop def\n";
to << "/vertoffset 0 descender sub printSize sub printFont /Courier ne\n";
to << "printFont /Courier-Bold ne and { 1 add } if def {\n";
to << "0 vertoffset moveto show\n";
to << "/vertoffset vertoffset printSize sub def\n";
to << "} forall\n";
to << "grestore\n";
to << "end\n";
to << "} dup 0 3 dict put def\n\n";
to << "/patternproc {\n";
to << "0 begin\n";
to << "/patternByteLength patternString length def\n";
to << "/patternHeight patternByteLength 8 mul sqrt cvi def\n";
to << "/patternWidth patternHeight def\n";
to << "/patternByteWidth patternWidth 8 idiv def\n";
to << "/imageByteMaxLength imageByteWidth imageHeight mul\n";
to << "stringLimit patternByteWidth sub min def\n";
to << "/imageMaxHeight imageByteMaxLength imageByteWidth idiv ";
to << "patternHeight idiv\n";
to << "patternHeight mul patternHeight max def\n";
to << "/imageHeight imageHeight imageMaxHeight sub store\n";
to << "/imageString imageByteWidth imageMaxHeight mul patternByteWidth ";
to << "add string def\n";
to << "0 1 imageMaxHeight 1 sub {\n";
to << "/y exch def\n";
to << "/patternRow y patternByteWidth mul patternByteLength mod def\n";
to << "/patternRowString patternString patternRow patternByteWidth ";
to << "getinterval def\n";
to << "/imageRow y imageByteWidth mul def\n";
to << "0 patternByteWidth imageByteWidth 1 sub {\n";
to << "/x exch def\n";
to << "imageString imageRow x add patternRowString putinterval\n";
to << "} for\n";
to << "} for\n";
to << "imageString\n";
to << "end\n";
to << "} dup 0 12 dict put def\n\n";
to << "/min {\n";
to << "dup 3 2 roll dup 4 3 roll lt { exch } if pop\n";
to << "} def\n\n";
to << "/max {\n";
to << "dup 3 2 roll dup 4 3 roll gt { exch } if pop\n";
to << "} def\n\n";
to << "/arrowhead {\n";
to << "0 begin\n";
to << "transform originalCTM itransform\n";
to << "/taily exch def\n";
to << "/tailx exch def\n";
to << "transform originalCTM itransform\n";
to << "/tipy exch def\n";
to << "/tipx exch def\n";
to << "/dy tipy taily sub def\n";
to << "/dx tipx tailx sub def\n";
to << "/angle dx 0 ne dy 0 ne or { dy dx atan } { 90 } ifelse def\n";
to << "gsave\n";
to << "originalCTM setmatrix\n";
to << "tipx tipy translate\n";
to << "angle rotate\n";
to << "newpath\n";
to << "0 0 moveto\n";
```

```
to << "arrowHeight neg arrowWidth 2 div lineto\n";
to << "arrowHeight neg arrowWidth 2 div neg lineto\n";
to << "closepath\n";
to << "patternNone not {\n";
to << "originalCTM setmatrix\n";
to << "/padtip arrowHeight 2 exp 0.25 arrowWidth 2 exp mul add sqrt ";
to << "brushWidth mul\n";
to << "arrowWidth div def\n";
to << "/padtail brushWidth 2 div def\n";
to << "tipx tipy translate\n";
to << "angle rotate\n";
to << "padtip 0 translate\n";
to << "arrowHeight padtip add padtail add arrowHeight div dup scale\n";
to << "arrowheadpath\n";
to << "ifill\n";
to << "} if\n";
to << "brushNone not {\n";
to << "originalCTM setmatrix\n";
to << "tipx tipy translate\n";
to << "angle rotate\n";
to << "arrowheadpath\n";
to << "istroke\n";
to << "} if\n";
to << "grestore\n";
to << "end\n";
to << "} dup 0 9 dict put def\n\n";
to << "/arrowheadpath {\n";
to << "newpath\n";
to << "0 0 moveto\n";
to << "arrowHeight neg arrowWidth 2 div lineto\n";
to << "arrowHeight neg arrowWidth 2 div neg lineto\n";
to << "closepath\n";
to << "} def\n\n";
to << "/leftarrow {\n";
to << "0 begin\n";
to << "y exch get /taily exch def\n";
to << "x exch get /tailx exch def\n";
to << "y exch get /tipy exch def\n";
to << "x exch get /tipx exch def\n";
to << "brushLeftArrow { tipx tipy tailx taily arrowhead } if\n";
to << "end\n";
to << "} dup 0 4 dict put def\n\n";
to << "/rightarrow {\n";
to << "0 begin\n";
to << "y exch get /tipy exch def\n";
to << "x exch get /tipx exch def\n";
to << "y exch get /taily exch def\n";
to << "x exch get /tailx exch def\n";
to << "brushRightArrow { tipx tipy tailx taily arrowhead } if\n";
to << "end\n";
to << "} dup 0 4 dict put def\n\n";
to << "/midpoint {\n";
to << "0 begin\n";
to << "/y1 exch def\n";
to << "/x1 exch def\n";
to << "/y0 exch def\n";
to << "/x0 exch def\n";
to << "x0 x1 add 2 div\n";
to << "y0 y1 add 2 div\n";
to << "end\n";
to << "} dup 0 4 dict put def\n\n";
to << "/thirdpoint {\n";
to << "0 begin\n";
to << "/y1 exch def\n";
to << "/x1 exch def\n";
to << "/y0 exch def\n";
to << "/x0 exch def\n";
to << "x0 2 mul x1 add 3 div\n";
to << "y0 2 mul y1 add 3 div\n";
to << "end\n";
to << "} dup 0 4 dict put def\n\n";
to << "/subspline {\n";
to << "0 begin\n";
to << "/movetoNeeded exch def\n";
to << "y exch get /y3 exch def\n";
to << "x exch get /x3 exch def\n";
```

```
    to << "y exch get /y2 exch def\n";
    to << "x exch get /x2 exch def\n";
    to << "y exch get /y1 exch def\n";
    to << "x exch get /x1 exch def\n";
    to << "y exch get /y0 exch def\n";
    to << "x exch get /x0 exch def\n";
    to << "x1 y1 x2 y2 thirdpoint\n";
    to << "/p1y exch def\n";
    to << "/p1x exch def\n";
    to << "x2 y2 x1 y1 thirdpoint\n";
    to << "/p2y exch def\n";
    to << "/p2x exch def\n";
    to << "x1 y1 x0 y0 thirdpoint\n";
    to << "p1x p1y midpoint\n";
    to << "/p0y exch def\n";
    to << "/p0x exch def\n";
    to << "x2 y2 x3 y3 thirdpoint\n";
    to << "p2x p2y midpoint\n";
    to << "/p3y exch def\n";
    to << "/p3x exch def\n";
    to << "movetoNeeded { p0x p0y moveto } if\n";
    to << "p1x p1y p2x p2y p3x p3y curveto\n";
    to << "end\n";
    to << "} dup 0 17 dict put def\n\n";
    to << "/storexyn {\n";
    to << "/n exch def\n";
    to << "/y n array def\n";
    to << "/x n array def\n";
    to << "n 1 sub -1 0 {\n";
    to << "/i exch def\n";
    to << "y i 3 2 roll put\n";
    to << "x i 3 2 roll put\n";
    to << "} for\n";
    to << "} def\n\n";
    to << "%%EndProlog\n\n";
}

// WriteDrawing writes code to store the picture's transformation
// matrix in a Postscript variable and code to draw the picture.

void PictSelection::WriteDrawing (ostream& to) {
    to << "\n\n%%Page: 1 1\n\n";
    to << "Begin\n";
    WritePictGS(to);
    to << "/originalCTM matrix currentmatrix def\n\n";

    for (First(); !AtEnd(); Next()) {
Selection* s = GetCurrent();
s->WriteData(to);
    }

    to << "End " << startdata << " eop\n\n";
    to << "showpage\n\n";
}

// WriteData writes the PictSelection's data and its children
// Selections' data with Postscript code to draw them.

void PictSelection::WriteData (ostream& to) {
    to << "Begin " << startdata << " Pict\n";
    WritePictGS(to);
    to << "\n";

    for (First(); !AtEnd(); Next()) {
Selection* s = GetCurrent();
s->WriteData(to);
    }

    to << "End " << startdata << " eop\n\n";
}

// WriteTrailer writes clean up code.

void PictSelection::WriteTrailer (ostream& to) {
    to << "%%Trailer\n\n";
    to << "end\n";
```

```
}

// ScaleToPostscriptCoords scales the picture to Postscript
// coordinates if screen and Postscript inches are different.

void PictSelection::ScaleToPostscriptCoords () {
   const double postscriptinch = 72.;

   if (inch != postscriptinch) {
double topostscript = postscriptinch / inch;
Scale(topostscript, topostscript);
   }
}

// ScaleToScreenCoords scales the picture back to screen coordinates
// if screen and Postscript inches are different.

void PictSelection::ScaleToScreenCoords () {
   const double postscriptinch = 72.;

   if (inch != postscriptinch) {
double toscreen = inch / postscriptinch;
Scale(toscreen, toscreen);
   }
}

// CollectFonts adds its font, if it has one, to the list without
// checking if the PictSelection contains any TextSelections. If it
// doesn't have a font, it collects its children TextSelection's
// fonts.

void PictSelection::CollectFonts (IFontList* fontlist) {
   if (GetFont() != nil) {
Merge((IFont*) GetFont(), fontlist);
   } else {
for (First(); !AtEnd(); Next()) {
   Selection* s = GetCurrent();
   if (s->HasChildren()) {
((PictSelection*) s)->CollectFonts(fontlist);
   } else if (s->IsA(TEXTSELECTION)) {
Merge((IFont*) s->GetFont(), fontlist);
   }
}
   }
}

// Merge merges the print font with the list of all print fonts unless
// the list already has it.

void PictSelection::Merge (IFont* font, IFontList* fontlist) {
   boolean found = false;
   if (font == nil) {
found = true;
   } else if (fontlist->Find(font)) {
found = true;
   } else {
for (fontlist->First(); !fontlist->AtEnd(); fontlist->Next()) {
   IFont* cmp = fontlist->GetCur()->GetFont();
   if (strcmp(font->GetPrintFont(), cmp->GetPrintFont()) == 0) {
found = true;
break;
   }
}
   }
   if (!found) {
fontlist->Append(new IFontNode(font));
   }
}

int PictSelection::FindIndex(Selection* s) {
int index = 0;
for (First(); !AtEnd(); Next(), ++index) {
if (GetCurrent() == s) {
return index;
      }
   }
```

```
return -1;
}

Selection* PictSelection::GetSelection(int index) {
  int counter = 0;
  for (First(); !AtEnd(); Next()) {
  if (counter == index) {
  return GetCurrent();
      }
++counter;
   }
return nil;
}
```

slpolygons.h

```
#ifndef slpolygons_h
#define slpolygons_h

#include "selection.h"

// A RectSelection draws a rectangle with an outline and a filled
// interior.

class RectSelection : public NPtSelection {
public:

   RectSelection(Coord, Coord, Coord, Coord, Graphic* = nil);
   RectSelection(istream&, State*);

   Graphic* Copy();
   void GetOriginal2(Coord&, Coord&, Coord&, Coord&);
   int GetOriginal(const Coord*&, const Coord*&);

protected:

   void WriteData(ostream&);
   RubberVertex* CreateRubberVertex(Coord*, Coord*, int, int);
   Selection* CreateReshapedCopy(Coord*, Coord*, int);

};

// A PolygonSelection draws a polygon with an outline and a filled
// interior.

class PolygonSelection : public NPtSelection {
public:

   PolygonSelection(Coord*, Coord*, int, Graphic* = nil);
   PolygonSelection(istream&, State*);

   Graphic* Copy();
   int GetOriginal(const Coord*&, const Coord*&);

protected:

   RubberVertex* CreateRubberVertex(Coord*, Coord*, int, int);
   Selection* CreateReshapedCopy(Coord*, Coord*, int);

};

#endif
```

slpolygons.c

```
#include "ipolygons.h"
#include "rubbands.h"
#include "slpolygons.h"
#include "dfdclasses.h"
#include <stream.h>

// RectSelection creates the rectangle's filled interior and outline.

RectSelection::RectSelection (Coord l, Coord b, Coord r, Coord t, Graphic* gs)
: (gs) {
```

```
    myname = "Rect";
    Append(new IFillRect(l, b, r, t));
    Append(new Rect(l, b, r, t));
    SetClassId(TERMINATOR);
}

// RectSelection reads data to initialize its graphic state and create
// its filled interior and outline.

RectSelection::RectSelection (istream& from, State* state) : (nil) {
    myname = "Rect";
    ReadGS(from, state);
    Skip(from);
    Coord l, b, r, t;
    from >> l >> b >> r >> t;
    Append(new IFillRect(l, b, r, t));
    Append(new Rect(l, b, r, t));
    SetClassId(TERMINATOR);
}

// Copy returns a copy of the RectSelection.

Graphic* RectSelection::Copy () {
    Coord l, b, r, t;
    GetOriginal2(l, b, r, t);
    return new RectSelection(l, b, r, t, this);
}

// GetOriginal2 returns the two corners that were passed to the
// RectSelection's constructor.

void RectSelection::GetOriginal2 (Coord& l, Coord& b, Coord& r, Coord& t) {
    ((Rect*) Last())->GetOriginal(l, b, r, t);
}

// GetOriginal returns the two corners that were passed to the
// RectSelection's constructor plus the other two opposite corners.

int RectSelection::GetOriginal (const Coord*& x, const Coord*& y) {
    static Coord sx[4], sy[4];
    GetOriginal2(sx[0], sy[0], sx[2], sy[2]);
    sx[1] = sx[0];
    sy[1] = sy[2];
    sx[3] = sx[2];
    sy[3] = sy[0];
    x = sx;
    y = sy;
    return 4;
}

// WriteData writes the RectSelection's data and Postscript code to
// draw it.

void RectSelection::WriteData (ostream& to) {
    Coord l, b, r, t;
    GetOriginal2(l, b, r, t);
    to << "Begin " << startdata << " Rect\n";
    WriteGS(to);
    to << startdata << "\n";
    to << l << " " << b << " " << r << " " << t << " Rect\n";
    to << "End\n\n";
}

// CreateRubberVertex creates and returns the right kind of
// RubberVertex to represent the RectSelection's shape.

RubberVertex* RectSelection::CreateRubberVertex (Coord* x, Coord* y,
int n, int rubpt) {
    return new RubberPolygon(nil, nil, x, y, n, rubpt);
}

// CreateReshapedCopy creates and returns a reshaped copy of itself
// using the passed points and its graphic state. It returns a
// PolygonSelection because the points may not shape a rect any more.

Selection* RectSelection::CreateReshapedCopy (Coord* x, Coord* y, int n) {
```

```
    return new PolygonSelection(x, y, n, this);
}

// PolygonSelection creates the polygon's filled interior and outline.

PolygonSelection::PolygonSelection (Coord* x, Coord* y, int n, Graphic* gs)
: (gs) {
    myname = "Poly";
    Append(new IFillPolygon(x, y, n));
    Append(new Polygon(x, y, n));
}

// PolygonSelection reads data to initialize its graphic state and
// create its filled interior and outline.

PolygonSelection::PolygonSelection (istream& from, State* state) : (nil) {
    myname = "Poly";
    ReadGS(from, state);
    Coord* x;
    Coord* y;
    int n;
    ReadPoints(from, x, y, n);
    Append(new IFillPolygon(x, y, n));
    Append(new Polygon(x, y, n));
}

// Copy returns a copy of the PolygonSelection.

Graphic* PolygonSelection::Copy () {
    Coord* x;
    Coord* y;
    int n = GetOriginal(x, y);
    Graphic* copy = new PolygonSelection(x, y, n, this);
    return copy;
}

// GetOriginal returns the vertices that were passed to the
// PolygonSelection's constructor.

int PolygonSelection::GetOriginal (const Coord*& x, const Coord*& y) {
    return ((Polygon*) Last())->GetOriginal(x, y);
}

// CreateRubberVertex creates and returns the right kind of
// RubberVertex to represent the PolygonSelection's shape.

RubberVertex* PolygonSelection::CreateRubberVertex (Coord* x, Coord* y,
int n, int rubpt) {
    return new RubberPolygon(nil, nil, x, y, n, rubpt);
}

// CreateReshapedCopy creates and returns a reshaped copy of itself
// using the passed points and its graphic state.

Selection* PolygonSelection::CreateReshapedCopy (Coord* x, Coord* y, int n) {
    return new PolygonSelection(x, y, n, this);
}
```

slsplines.h

```
#ifndef slsplines_h
#define slsplines_h

#include "selection.h"
#include "InterViews/defs.h"

class Edge;

// A BSplineSelection draws an open B-spline with a filled interior.

class BSplineSelection : public NPtSelection {
public:

    BSplineSelection(ClassId, Coord*, Coord*, int, Graphic* = nil);
    BSplineSelection(Coord*, Coord*, int, Graphic* = nil);
    BSplineSelection(istream&, State*);
```

```
    Graphic* Copy();
    int GetOriginal(const Coord*&, const Coord*&);
Selection* CreateReshapedCopy(Coord*, Coord*, int, ClassId);
    Selection* CreateReshapedCopy(Coord*, Coord*, int);
boolean IsAStream();
void SetStream();

protected:

    void Init(Coord*, Coord*, int);
    RubberVertex* CreateRubberVertex(Coord*, Coord*, int, int);

    void uncacheChildren();
    void getExtent(float&, float&, float&, float&, float&, Graphic*);
    boolean contains(PointObj&, Graphic*);
    boolean intersects(BoxObj&, Graphic*);
    void draw(Canvas*, Graphic*);
    void drawClipped(Canvas*, Coord, Coord, Coord, Coord, Graphic*);

    Graphic* ifillbspline;// fills an open B-spline
    Graphic* bspline;// draws an open B-spline
    Coord lx0, ly0, lx1, ly1;// stores endpoints of left arrowhead
    Coord rx0, ry0, rx1, ry1;// stores endpoints of right arrowhead
boolean is_stream;  // stores where data flow is stream or not
};

// A ClosedBSplineSelection draws a closed B-spline with a filled
// interior.

class ClosedBSplineSelection : public NPtSelection {
public:

    ClosedBSplineSelection(Coord*, Coord*, int, Graphic* = nil);
    ClosedBSplineSelection(istream&, State*);

    Graphic* Copy();
    int GetOriginal(const Coord*&, const Coord*&);

protected:

    RubberVertex* CreateRubberVertex(Coord*, Coord*, int, int);
    Selection* CreateReshapedCopy(Coord*, Coord*, int);

};

#endif
```

slsplines.c

```
#include "dfdclasses.h"
#include "isplines.h"
#include "slsplines.h"
#include <InterViews/rubcurve.h>
#include <InterViews/Std/stream.h>

// BSplineSelection creates its components.

BSplineSelection::BSplineSelection (ClassId classid, Coord* x, Coord* y,
      int n,Graphic* gs) : (classid, gs) {
    Init(x, y, n);
}

BSplineSelection::BSplineSelection (Coord* x, Coord* y, int n, Graphic* gs)
: (gs) {
    Init(x, y, n);
}

// BSplineSelection reads data to initialize its graphic state and
// create its components.

BSplineSelection::BSplineSelection (istream& from, State* state)
: (nil) {
    bspline = nil;
    ReadGS(from, state);
```

```
    Coord* x;
    Coord* y;
    int n;
    ReadPoints(from, x, y, n);
    Init(x, y, n);
}

// Copy returns a copy of the BSplineSelection.

Graphic* BSplineSelection::Copy () {
    Coord* x;
    Coord* y;
    int n = GetOriginal(x, y);
    Graphic* copy = new BSplineSelection(x, y, n, this);
    return copy;
}

// GetOriginal returns the control points that were passed to the
// BSplineSelection's constructor.

int BSplineSelection::GetOriginal (const Coord*& x, const Coord*& y) {
    return ((BSpline*) bspline)->GetOriginal(x, y);
}

// Init creates the graphic's components and stores the arrowheads'
// endpoints.

void BSplineSelection::Init (Coord* x, Coord* y, int n) {
    is_stream = false;

    myname = "BSpl";
    ifillbspline = new IFillBSpline(x, y, n);
    bspline = new BSpline(x, y, n);
    lx0 = x[0];
    ly0 = y[0];
    lx1 = x[1];
    ly1 = y[1];
    rx0 = x[n-1];
    ry0 = y[n-1];
    rx1 = x[n-2];
    ry1 = y[n-2];
}

// CreateRubberVertex creates and returns the right kind of
// RubberVertex to represent the BSplineSelection's shape.

RubberVertex* BSplineSelection::CreateRubberVertex (Coord* x, Coord* y,
int n, int rubpt) {
    return new RubberSpline(nil, nil, x, y, n, rubpt);
}

// CreateReshapedCopy creates and returns a reshaped copy of itself
// using the passed points and its graphic state.

Selection* BSplineSelection::CreateReshapedCopy (Coord* x, Coord* y, int n,
 ClassId cid) {
    return new BSplineSelection(cid, x, y, n, this);
}

Selection* BSplineSelection::CreateReshapedCopy (Coord* x, Coord* y, int n) {
return new BSplineSelection(cid, x, y, n, this);
}

// uncacheChildren uncaches the graphic's components' extents.

void BSplineSelection::uncacheChildren () {
    if (bspline != nil) {
uncacheExtentGraphic(ifillbspline);
uncacheExtentGraphic(bspline);
    }
}

// getExtent returns the graphic's extent including a tolerance for
// the arrowheads.

void BSplineSelection::getExtent (float& l, float& b, float& cx, float& cy,
```

```
float& tol, Graphic* gs) {
   Extent e;
   if (extentCached()) {
getCachedExtent(e.left, e.bottom, e.cx, e.cy, e.tol);
   } else {
FullGraphic gstmp;
concatGSGraphic(ifillbspline, this, gs, &gstmp);
getExtentGraphic(
        ifillbspline, e.left, e.bottom, e.cx, e.cy, e.tol, &gstmp
      );
Extent te;
concatGSGraphic(bspline, this, gs, &gstmp);
getExtentGraphic(
        bspline, te.left, te.bottom, te.cx, te.cy, te.tol, &gstmp
      );
e.Merge(te);
cacheExtent(e.left, e.bottom, e.cx, e.cy, e.tol);
   }
   float right = 2*e.cx - e.left;
   float top = 2*e.cy - e.bottom;
   float dummy1, dummy2;
   transformRect(e.left, e.bottom, right, top, l, b, dummy1, dummy2, gs);
   transform(e.cx, e.cy, cx, cy, gs);
   tol = MergeArrowHeadTol(e.tol, gs);
}

// contains returns true if the graphic contains the point.

boolean BSplineSelection::contains (PointObj& po, Graphic* gs) {
   BoxObj b;
   getBox(b, gs);
   if (b.Contains(po)) {
if (containsGraphic(ifillbspline, po, gs)) {
   return true;
} else if (containsGraphic(bspline, po, gs)) {
   return true;
} else if (LeftAcont(lx0, ly0, lx1, ly1, po, gs)) {
   return true;
} else if (RightAcont(rx0, ry0, rx1, ry1, po, gs)) {
   return true;
}
   }
   return false;
}

// intersects returns true if the graphic intersects the box.

boolean BSplineSelection::intersects (BoxObj& userb, Graphic* gs) {
   BoxObj b;
   getBox(b, gs);
   if (b.Intersects(userb)) {
if (intersectsGraphic(ifillbspline, userb, gs)) {
   return true;
} else if (intersectsGraphic(bspline, userb, gs)) {
   return true;
} else if (LeftAints(lx0, ly0, lx1, ly1, userb, gs)) {
   return true;
} else if (RightAints(rx0, ry0, rx1, ry1, userb, gs)) {
   return true;
}
   }
   return false;
}

// draw draws the graphic.

void BSplineSelection::draw (Canvas* c, Graphic* gs) {
   drawGraphic(ifillbspline, c, gs);
   drawGraphic(bspline, c, gs);
   drawLeftA(lx0, ly0, lx1, ly1, c, gs);
   drawRightA(rx0, ry0, rx1, ry1, c, gs);
}

// drawClipped draws the graphic if it intersects the clipping box.

void BSplineSelection::drawClipped (Canvas* c, Coord l, Coord b, Coord r,
```

```
Coord t, Graphic* gs) {
   BoxObj box;
   getBox(box, gs);

   BoxObj clipBox(l, b, r, t);
   if (clipBox.Intersects(box)) {
draw(c, gs);
   }
}

// ClosedBSplineSelection creates the closed B-spline's filled
// interior and outline.

ClosedBSplineSelection::ClosedBSplineSelection (Coord* x, Coord* y, int n,
Graphic* gs) : (gs) {
   myname = "CBSpl";
   Append(new IFillClosedBSpline(x, y, n));
   Append(new ClosedBSpline(x, y, n));
}

// ClosedBSplineSelection reads data to initialize its graphic state
// and create the closed B-spline's filled interior and outline.

ClosedBSplineSelection::ClosedBSplineSelection (istream& from, State* state)
: (nil) {
   myname = "CBSpl";
   ReadGS(from, state);
   Coord* x;
   Coord* y;
   int n;
   ReadPoints(from, x, y, n);
   Append(new IFillClosedBSpline(x, y, n));
   Append(new ClosedBSpline(x, y, n));
}

// Copy returns a copy of the ClosedBSplineSelection.

Graphic* ClosedBSplineSelection::Copy () {
   Coord* x;
   Coord* y;
   int n = GetOriginal(x, y);
   Graphic* copy = new ClosedBSplineSelection(x, y, n, this);
   return copy;
}

// GetOriginal returns the control points that were passed to the
// ClosedBSplineSelection's constructor.

int ClosedBSplineSelection::GetOriginal (const Coord*& x, const Coord*& y) {
   return ((ClosedBSpline*) Last())->GetOriginal(x, y);
}

// CreateRubberVertex creates and returns the right kind of
// RubberVertex to represent the ClosedBSplineSelection's shape.

RubberVertex* ClosedBSplineSelection::CreateRubberVertex (Coord* x, Coord* y,
int n, int rubpt) {
   return new RubberClosedSpline(nil, nil, x, y, n, rubpt);
}

// CreateReshapedCopy creates and returns a reshaped copy of itself
// using the passed points and its graphic state.

Selection* ClosedBSplineSelection::CreateReshapedCopy (Coord* x, Coord* y,
int n) {
   return new ClosedBSplineSelection(x, y, n, this);
}

// return whether spline is stream or not

boolean BSplineSelection::IsAStream() {
return is_stream;
}

// set value of is_stream
```

```
void BSplineSelection::SetStream() {
is_stream = true;
}
```

sltext.h

```
#ifndef sltext_h
#define sltext_h

#include "selection.h"

// Declare imported types.

class TextBuffer;

// A TextSelection draws one to several lines of text.

static const ClassId TEXTSELECTION = 2100;

class TextSelection : public Selection {
public:

TextSelection(ClassId, const char*, int, Graphic* = nil);
    TextSelection(const char*, int, Graphic* = nil);
    TextSelection(istream&, State*);
    ~TextSelection();

    Graphic* Copy();
    boolean IsA(ClassId);

    const char* GetOriginal(int&);
    boolean ShapedBy(Coord, Coord, float);

    char *GetString();
    char *GetValidString();

    boolean GetDefPosition();
    void SetDefPosition(boolean);

protected:

    void draw(Canvas*, Graphic*);
    void drawClipped(Canvas*, Coord, Coord, Coord, Coord, Graphic*);
    void ReadjustSpacing(PFont*);

    void Init(const char*, int);
    const char* ReadTextData(istream&, int&);
    void WriteData(ostream&);
    const char* Filter(const char*, int);

    boolean defPosition;
    char* tbuf;// stores the TextSelection's text
    TextBuffer* text;// operates on the text
    int lineHt;// stores previous line height

};

#endif
```

sltext.c

```
#include "dfdclasses.h"
#include "ipaint.h"
#include "istring.h"
#include "sltext.h"
#include <InterViews/Graphic/label.h>
#include <InterViews/textbuffer.h>
#include <bstring.h>
#include <stream.h>

// Both ReadTextData and Filter use this big static buffer.

static const int SBUFSIZE = 10000;
static char sbuf[SBUFSIZE];
```

```
// TextSelection gets passed its graphic state, text, and type of text.

TextSelection::TextSelection (ClassId classid, const char* orig, int len,
Graphic* gs) : (classid, gs) {
    Init(orig, len);
}

// TextSelection gets passed its graphic state and text.

TextSelection::TextSelection (const char* orig, int len, Graphic* gs)
 : (COMMENT, gs) {
    Init(orig, len);
}

// TextSelection reads its graphic state and text from a file.

TextSelection::TextSelection (istream& from, State* state) :
        Selection((Graphic*)nil) {
    ReadTextGS(from, state);
    int len;
    const char* orig = ReadTextData(from, len);
    Init(orig, len);
}

// Init copies the original text and appends Labels for each line of
// text to the TextSelection.

void TextSelection::Init (const char* orig, int len) {
    defPosition = true;

    int tbufsize = max(len, 1);
    tbuf = new char[tbufsize];
    bcopy(orig, tbuf, len);
    text = new TextBuffer(tbuf, len, tbufsize);
    lineHt = 0;

    int lines = text->Height();
    for (int i = 0; i < lines; ++i) {
int bol = text->LineIndex(i);
int eol = text->EndOfLine(bol);
/* need this crock for now because Label(t, 0) is buggy */
if (eol - bol > 0) {
    Append(new Label(text->Text(bol, eol), eol - bol));
} else {
    Append(new Label(""));
}
    }
    ReadjustSpacing(GetFont());
}

// Free storage allocated for the text in the TextSelection.

TextSelection::~TextSelection () {
    delete text;
    delete tbuf;
}

// Copy returns a copy of the TextSelection.

Graphic* TextSelection::Copy () {
    return new TextSelection(GetClassId(), text->Text(), text->Length(), this);
}

// IsA returns true if the TextSelection is a TextSelection so Editor
// can identify TextSelections and edit them differently.

boolean TextSelection::IsA (ClassId id) {
    return id == LABEL_OP || id == LABEL_DF || id == LABEL_SL ||
 id == COMMENT || id == MET_OP || id == LAT_DF;
}

// GetOriginal returns a pointer to the TextSelection's text.

const char* TextSelection::GetOriginal (int& len) {
    len = text->Length();
    return text->Text();
```

```
}

char *TextSelection::GetString() {
  int len;
  const char *str = GetOriginal(len);
  char *rval = new char[len + 1];
  strncpy(rval,str,len);
  rval[len] = 0;
  return rval;
}

char *TextSelection::GetValidString() {
  char *str = GetString();
  char *rval = RemoveBadChars(str);
  delete [] str;
  return rval;
}

// ShapedBy returns true if the TextSelection intersects a box around
// the given point.

boolean TextSelection::ShapedBy (Coord px, Coord py, float maxdist) {
   int slop = round(maxdist / 2);
   BoxObj pickpoint(px - slop, py - slop, px + slop, py + slop);
   return (LastGraphicIntersecting(pickpoint) != nil);
}

// draw readjusts the spacing between lines of text for a new font if
// necessary and sets fillbg false and pattern solid to draw the text
// like the printer will.

void TextSelection::draw (Canvas* c, Graphic* gs) {
   ReadjustSpacing(gs->GetFont());
   boolean fillbg = gs->BgFilled();
   PPattern* pattern = gs->GetPattern();

   gs->SetPattern(psolid);
   gs->FillBg(false);
   Selection::draw(c, gs);
   gs->FillBg(fillbg);
   gs->SetPattern(pattern);
}

// drawClipped readjusts the spacing between lines of text for a new
// font if necessary and sets fillbg false and pattern solid to draw
// the text like the printer will.

void TextSelection::drawClipped (Canvas* c, Coord l, Coord b, Coord r, Coord t,
Graphic* gs) {
   ReadjustSpacing(gs->GetFont());
   boolean fillbg = gs->BgFilled();
   PPattern* pattern = gs->GetPattern();

   gs->SetPattern(psolid);
   gs->FillBg(false);
   Selection::drawClipped(c, l, b, r, t, gs);
   gs->FillBg(fillbg);
   gs->SetPattern(pattern);
}

// ReadjustSpacing recalculates the spacing between lines of text.

void TextSelection::ReadjustSpacing (PFont* font) {
   if (font != nil) {
int newHt = ((IFont*) font)->GetLineHt();
if (lineHt != newHt) {
   lineHt = newHt;
   int vertoffset = -lineHt + 1;
   for (First(); !AtEnd(); Next()) {
Graphic* label = GetCurrent();
label->SetTransformer(nil);
label->Translate(0, vertoffset);
vertoffset -= lineHt;
   }
}
   }
```

```
}

// ReadTextData reads and returns the text contained in the PostScript
// representation of the TextSelection.

const char* TextSelection::ReadTextData (istream& from, int& len) {
    TextBuffer stext(sbuf, 0, SBUFSIZE);
    char nl = '\n';

    if (versionnumber >= NONREDUNDANTVERSION) {
Skip(from);
int dot = 0;
char c = ' ';
while (from >> c && c != ']') {
    while (c != '(' && from.get(c)) {
    }
    while (from.get(c) && c != ')') {
if (c == '\\') {
    from.get(c);
}
stext.Insert(dot++, &c, 1);
    }
    stext.Insert(dot++, &nl, 1);
}
stext.Delete(--dot, 1); // buffer must not terminate in '\n'
    } else {
int dot = 0;
while (from >> buf && strcmp(buf, startdata) == 0) {
    char blank;
    from.get(blank);
    from.get(buf, BUFSIZE - 1);
    int buflen = strlen(buf) + 1;
    buf[buflen - 1] = '\n';
    stext.Insert(dot, buf, buflen);
    dot += buflen;
}
stext.Delete(--dot, 1); // buffer must not terminate in '\n'
    }

    len = stext.Length();
    return stext.Text();
}

// WriteData writes the TextSelection's data and Postscript code to
// draw it.

void TextSelection::WriteData (ostream& to) {
    to << "Begin " << startdata << " Text\n";
    WriteTextGS(to);
    to << startdata << "\n";
    to << "[\n";

    int lines = text->Height();
    for (int i = 0; i < lines; ++i) {
int bol = text->LineIndex(i);
int eol = text->EndOfLine(bol);
const char* string = Filter(text->Text(bol, eol), eol - bol);
to << "(" << string << ")\n";
    }

    to << "] Text\n";
    to << "End\n\n";
}

// Filter escapes embedded special characters that would cause syntax
// errors in a Postscript string.

const char* TextSelection::Filter (const char* string, int len) {
    TextBuffer stext(sbuf, 0, SBUFSIZE);
    char esc = '\\';
    char nul = '\0';

    int dot = 0;
    for (int i = 0; i < len; i++) {
switch (*string) {
case '(':
```

```
case ')':
case '\\':
  stext.Insert(dot++, &esc, 1);
  // fall through
default:
  stext.Insert(dot++, string++, 1);
}
  }
  stext.Insert(dot++, &nul, 1);

  return stext.Text();
}

boolean TextSelection::GetDefPosition() {
  return defPosition;
}

void TextSelection::SetDefPosition(boolean d) {
  defPosition = d;
}
```

state.h

```
#ifndef state_h
#define state_h

#include <InterViews/defs.h>

// Declare imported types.

class Graphic;
class IBrush;
class IColor;
class IFont;
class IPattern;
class Interactor;
class InteractorList;
class MapIBrush;
class MapIColor;
class MapIFont;
class MapIPattern;
class Page;
class Transformer;

// A State stores state information about the user's drawing and paint
// attributes to be used when creating new Selections.

enum ModifStatus {
  ReadOnly,// no modifications to drawing allowed
  Unmodified,// no modifications have been made yet
  Modified// at least one modification has been made
};

class State {
public:

  State(Interactor*, Page*);
  ~State();

  void Constrain(Coord&, Coord&);
  void ToggleOrientation();

  IBrush* GetBrush();
  IColor* GetFgColor();
  int GetFgColorIndex(IColor*);
  IColor* GetBgColor();
  int GetBgColorIndex(IColor*);
  const char* GetDrawingName();
  const char* GetMessage();
  boolean GetFillBg();
  IFont* GetFont();
  int GetFontIndex(IFont*);
  Graphic* GetGraphicGS();
  boolean GetGridGravity();
  double GetGridSpacing();
  boolean GetGridVisibility();
```

```
    int GetLineHt();
    float GetMagnif();
    MapIBrush* GetMapIBrush();
    MapIColor* GetMapIFgColor();
    MapIColor* GetMapIBgColor();
    MapIFont* GetMapIFont();
    MapIPattern* GetMapIPattern();
    ModifStatus GetModifStatus();
    IPattern* GetPattern();
    IPattern* GetArrowPattern();
    Graphic* GetTextGS();
    Painter* GetTextPainter();

    void SetBrush(IBrush*);
    void SetFgColor(IColor*);
    void SetBgColor(IColor*);
    void SetDrawingName(const char*);
    void SetMessage(const char*);
    void SetFillBg(boolean);
    void SetFont(IFont*);
    void SetGraphicT(Transformer&);
    void SetGridGravity(boolean);
    void SetGridSpacing(double);
    void SetGridVisibility(boolean);
    void SetMagnif(float);
    void SetModifStatus(ModifStatus);
    void SetPattern(IPattern*);
    void SetTextGS(Coord, Coord, Painter*);
    void SetTextGS(Graphic*, Painter*);

    void Attach(Interactor*);
    void Detach(Interactor*);
    void UpdateViews();

protected:

    char* drawingname;// stores drawing's filename
    char* msg;          // stores editor message string
    Graphic* graphicstate;// stores all attributes for creating graphics
    Transformer* graphicstate_t;// stores matrix for creating graphics
    boolean gridding;// stores true if grid will constrain points
    double gridspacing;// stores spacing between grid pts in points
    float magnif;// stores drawing view's magnification factor
    MapIBrush* mapibrush;// stores list of possible brushes
    MapIColor* mapifgcolor;// stores list of possible fg colors
    MapIColor* mapibgcolor;// stores list of possible bg colors
    MapIFont* mapifont;// stores list of possible fonts
    MapIPattern* mapipattern;// stores list of possible patterns
    MapIPattern* mapiarrowpattern; // stores list of possible arrow patterns
    ModifStatus modifstatus;// stores drawing's modification status
    Page* page;// stores all grid attributes
    Graphic* textgs;// stores all attributes for creating text
    Transformer* textgs_t;// stores matrix for creating text
    Painter* textpainter;// stores all attributes for editing text
    Transformer* textpainter_t;// stores matrix for editing text
    float textx;// stores last place text was being edited
    float texty;// stores last place text was being edited
    int lineHt;// stores spacing between lines of text
    InteractorList* viewlist;// stores list of views to notify

};

#endif
```

state.c

```
#include "ipaint.h"
#include "istring.h"
#include "listintrctr.h"
#include "mapipaint.h"
#include "page.h"
#include "state.h"
#include <InterViews/graphic.h>
#include <InterViews/interactor.h>
#include <InterViews/painter.h>
#include <InterViews/transformer.h>
```

```
// State stores some Graphic and nonGraphic attributes.

State::State (Interactor* i, Page* p) {
    drawingname = nil;
msg = nil;
    graphicstate = new FullGraphic;
    graphicstate_t = new Transformer;
    magnif = 1.0;
    mapibrush = new MapIBrush(i, "brush");
    mapifgcolor = new MapIColor(i, "fgcolor");
    mapibgcolor = new MapIColor(i, "bgcolor");
    mapifont = new MapIFont(i, "font");
    mapipattern = new MapIPattern(i, "pattern");
mapiarrowpattern = new MapIPattern(i, "arrowpattern");
    modifstatus = Unmodified;
    page = p;
    textgs = new FullGraphic;
    textgs_t = new Transformer;
    textpainter = new Painter;
    textpainter->Reference();
    textpainter_t = new Transformer;
    viewlist = new InteractorList;

    graphicstate->SetBrush(mapibrush->GetInitial());
    graphicstate->SetColors(
  mapifgcolor->GetInitial(), mapibgcolor->GetInitial()
    );
    graphicstate->FillBg(true);
    graphicstate->SetFont(mapifont->GetInitial());
    graphicstate->SetPattern(mapipattern->GetInitial());
    graphicstate->SetTransformer(graphicstate_t);
    textgs->SetTransformer(textgs_t);
    textpainter->SetTransformer(textpainter_t);
}

// ~State frees storage allocated to store members.

State::~State () {
    delete drawingname;
    delete graphicstate;
    delete mapibrush;
    delete mapifgcolor;
    delete mapibgcolor;
    delete mapifont;
    delete mapipattern;
delete mapiarrowpattern;
    delete textgs;
    Unref(textpainter);
    delete viewlist;
}

// The following functions add Page operations to State.

void State::Constrain (Coord& x, Coord& y) {
    page->Constrain(x, y);
}

void State::ToggleOrientation () {
    page->ToggleOrientation();
}

// The following functions return Graphic and nonGraphic attributes of
// the State.

IBrush* State::GetBrush () {
    return (IBrush*) graphicstate->GetBrush();
}

IColor* State::GetFgColor () {
    return (IColor*) graphicstate->GetFgColor();
}

IColor* State::GetBgColor () {
    return (IColor*) graphicstate->GetBgColor();
}
```

```
int State::GetFgColorIndex (IColor *c) {
  int count = 0;
  for (mapifgcolor->First();!mapifgcolor->AtEnd();mapifgcolor->Next()) {
    if (mapifgcolor->GetCur() == c)
      return count;
    count++;
  }
  return -1;
}

int State::GetBgColorIndex (IColor *c) {
  int count = 0;
  for (mapibgcolor->First();!mapibgcolor->AtEnd();mapibgcolor->Next()) {
    if (mapibgcolor->GetCur() == c)
      return count;
    count++;
  }
  return -1;
}

const char* State::GetDrawingName () {
  return drawingname;
}

const char* State::GetMessage() {
return msg;
}

boolean State::GetFillBg () {
  return graphicstate->BgFilled();
}

IFont* State::GetFont () {
  return (IFont*) graphicstate->GetFont();
}

int State::GetFontIndex (IFont *f) {
  int count = 0;
  for (mapifont->First();!mapifont->AtEnd();mapifont->Next()) {
    if (mapifont->GetCur() == f)
      return count;
    count++;
  }
  return -1;
}

Graphic* State::GetGraphicGS () {
  return graphicstate;
}

boolean State::GetGridGravity () {
  return page->GetGridGravity();
}

double State::GetGridSpacing () {
  return page->GetGridSpacing();
}

boolean State::GetGridVisibility () {
  return page->GetGridVisibility();
}

int State::GetLineHt () {
  return lineHt;
}

float State::GetMagnif () {
  return magnif;
}

MapIBrush* State::GetMapIBrush () {
  return mapibrush;
}

MapIColor* State::GetMapIFgColor () {
```

```
    return mapifgcolor;
}

MapIColor* State::GetMapIBgColor () {
    return mapibgcolor;
}

MapIFont* State::GetMapIFont () {
    return mapifont;
}

MapIPattern* State::GetMapIPattern() {
 return mapipattern;
}

ModifStatus State::GetModifStatus () {
    return modifstatus;
}

// Return stored arrow pattern

IPattern* State::GetArrowPattern() {
 return mapiarrowpattern->GetCur();
}

IPattern* State::GetPattern () {
    return (IPattern*) graphicstate->GetPattern();
}

Graphic* State::GetTextGS () {
    return textgs;
}

Painter* State::GetTextPainter () {
    return textpainter;
}

// The following functions set Graphic and nonGraphic attributes of
// the State.

void State::SetBrush (IBrush* b) {
    graphicstate->SetBrush(b);
}

void State::SetFgColor (IColor* fg) {
    graphicstate->SetColors(fg, graphicstate->GetBgColor());
}

void State::SetBgColor (IColor* bg) {
    graphicstate->SetColors(graphicstate->GetFgColor(), bg);
}

void State::SetDrawingName (const char* name) {
    delete drawingname;
    drawingname = name ? strdup(name) : nil;
}

void State::SetMessage(const char* m) {
delete msg;
msg = m ? strdup(m) : nil;
}

void State::SetFillBg (boolean fill) {
    graphicstate->FillBg(fill);
}

void State::SetFont (IFont* f) {
    graphicstate->SetFont(f);
}

void State::SetGraphicT (Transformer& t) {
    *graphicstate_t = t;
    graphicstate_t->Invert();

    Transformer tnew;
    float left, top;
```

```
    graphicstate_t->InvTransform(textx, texty, left, top);
    tnew.Scale(magnif, magnif);
    tnew.Translate(left, top);
    *textpainter_t = tnew;
    tnew.Postmultiply(graphicstate_t);
    *textgs_t = tnew;
}

void State::SetGridGravity (boolean g) {
    page->SetGridGravity(g);
}

void State::SetGridSpacing (double s) {
    page->SetGridSpacing(s);
}

void State::SetGridVisibility (boolean v) {
    page->SetGridVisibility(v);
}

void State::SetMagnif (float m) {
    magnif = m;
}

void State::SetModifStatus (ModifStatus m) {
    modifstatus = m;
}

void State::SetPattern (IPattern* p) {
    graphicstate->SetPattern(p);
}

void State::SetTextGS (Coord left, Coord top, Painter* output) {
    PColor* fg = graphicstate->GetFgColor();
    PFont* f = graphicstate->GetFont();
    lineHt = ((IFont*) f)->GetLineHt();
    textgs->SetColors(fg, textgs->GetBgColor());
    textgs->SetFont(f);
    textpainter->SetColors(*fg, output->GetBgColor());
    textpainter->SetFont(*f);
    graphicstate_t->Transform(float(left), float(top), textx, texty);

    Transformer t;
    t.Scale(magnif, magnif);
    t.Translate(left, top);
    *textpainter_t = t;
    t.Postmultiply(graphicstate_t);
    *textgs_t = t;
}

void State::SetTextGS (Graphic* gs, Painter* output) {
    PColor* fg = gs->GetFgColor();
    PFont* f = gs->GetFont();
    lineHt = ((IFont*) f)->GetLineHt();
    textgs->SetColors(fg, textgs->GetBgColor());
    textgs->SetFont(f);
    textpainter->SetColors(*fg, output->GetBgColor());
    textpainter->SetFont(*f);
    gs->TotalTransformation(*textpainter_t);
    Transformer* t = gs->GetTransformer();
    if (t != nil) {
t->Transform(0.0, 0.0, textx, texty);
    }
}

// Attach informs us a view has attached itself to us.

void State::Attach (Interactor* i) {
    viewlist->Append(new InteractorNode(i));
}

// Detach informs us a view has detached itself from us.

void State::Detach (Interactor* i) {
    if (viewlist->Find(i)) {
viewlist->DeleteCur();
```

```
    }
}

// UpdateViews informs all attached views we have changed our state.

void State::UpdateViews () {
    for (viewlist->First(); !viewlist->AtEnd(); viewlist->Next()) {
Interactor* view = viewlist->GetCur()->GetInteractor();
view->Update();
    }
}
```

stateviews.h

```
#ifndef stateviews_h
#define stateviews_h

#include <InterViews/interactor.h>

// Declare imported types.

class Graphic;
class State;

// A StateView attaches itself to a State and displays some of the
// State's information.

class StateView : public Interactor {
public:

    StateView(State*, const char*);
    ~StateView();

protected:

    void Reconfig();
    void Redraw(Coord, Coord, Coord, Coord);
    void Resize();

    State* state;// stores subject whose attribute view displays
    char* label;// stores view's text label

    Coord label_x, label_y;// stores position at which to display label

};

// A BrushView displays the current brush.

class BrushView : public StateView {
public:

    BrushView(State*);
    ~BrushView();

    void Update();

protected:

    void Reconfig();
    void Redraw(Coord, Coord, Coord, Coord);
    void Resize();

    Graphic* brushindic;// displays line to demonstrate brush's effect

};

// A DrawingNameView displays the drawing's name.

class DrawingNameView : public StateView {
public:

    DrawingNameView(State*);

    void Update();

protected:
```

```
    void Reconfig();
    void Resize();
    const char* GetDrawingName(State*);

};

// A FontView displays the current font.

class FontView : public StateView {
public:

    FontView(State*);
    ~FontView();

    void Update();

protected:

    void Reconfig();
    void Redraw(Coord, Coord, Coord, Coord);
    const char* GetPrintFontAndSize(State*);

    Painter* background;// draws background behind label

};

// A GriddingView displays the current status of the grid's gridding.

class GriddingView : public StateView {
public:

    GriddingView(State*);

    void Update();

protected:

    const char* GetGridding(State*);

};

// A MagnifView displays the current magnification.

class MagnifView : public StateView {
public:

    MagnifView(State*, Interactor*);

    void Update();

protected:

    void Resize();
    const char* GetMagnif(State*);

};

// A ModifStatusView displays the current modification status.

class ModifStatusView : public StateView {
public:

    ModifStatusView(State*);

    void Update();

protected:

    const char* GetModifStatus(State*);

};

// A PatternView displays the current pattern.

class PatternView : public StateView {
```

```
public:

   PatternView(State*);
   ~PatternView();

   void Update();

protected:

   void Reconfig();
   void Redraw(Coord, Coord, Coord, Coord);

   Painter* patindic;// fills rect to demonstrate pat's effect

};

class MsgView : public StateView {
public:
MsgView(State*);
void Update();
protected:
void Reconfig();
void Resize();
const char* GetMessage(State*);
};

#endif
```

stateviews.c

```
#include "ipaint.h"
#include "istring.h"
#include "sllines.h"
#include "state.h"
#include "stateviews.h"
#include <InterViews/painter.h>
#include <InterViews/perspective.h>
#include <InterViews/shape.h>
#include <InterViews/Std/stdio.h>

// StateView attaches itself to the State's list of views to update
// and stores the State and text label.

StateView::StateView (State* s, const char* l) {
   s->Attach(this);
   state = s;
   label = strdup(l ? l : "");
}

// Free storage allocated for the text label.

StateView::~StateView () {
   delete label;
}

// Reconfig pads the view's shape to accomodate its text label.
// Basing padding on the font in use ensures the padding will change
// proportionally with changes in the font's size.

static const float WIDTHPAD = 1.0; // fraction of font->Width(EM)
static const float HTPAD = 0.2;   // fraction of font->Height()
static const char* EM = "m";   // widest alphabetic character in any font

void StateView::Reconfig () {
   Interactor::Reconfig();
   Font* font = output->GetFont();
   int xpad = round(WIDTHPAD * font->Width(EM));
   int ypad = round(HTPAD * font->Height());
   shape->width = font->Width(label) + (2 * xpad);
   shape->height = font->Height() + (2 * ypad);
   shape->Rigid(shape->width - xpad, 0, 2 * ypad, 0);
}

// Redraw displays the text label.

void StateView::Redraw (Coord l, Coord b, Coord r, Coord t) {
   output->ClearRect(canvas, l, b, r, t);
```

```
    output->Text(canvas, label, label_x, label_y);
}

// Resize centers the text label's position unless the label won't fit
// on the canvas, in which case Resize left justifies the position.

void StateView::Resize () {
    Font* font = output->GetFont();
    label_x = max(0, (xmax - font->Width(label) + 1) / 2);
    label_y = (ymax - font->Height() + 1) / 2;
}

// BrushView creates a graphic to demonstrate the brush's effect on a
// line.

static const int PICXMAX = 28;// chosen to minimize scaling for canvas
static const int PICYMAX = 18;

BrushView::BrushView (State* s) : (s, " N ") {

// store pattern in temporary variable

IPattern* temp = state->GetPattern();

// make pattern state to be the arrow pattern

state->SetPattern(state->GetArrowPattern());

// draw line with arrow pattern

    brushindic = new LineSelection(0, 0, PICXMAX, 0, state->GetGraphicGS());

// reset pattern

state->SetPattern(temp);
    brushindic->SetTransformer(nil);
}

// Free storage allocated for the graphic.

BrushView::~BrushView () {
    delete brushindic;
}

// Skew comments/code ratio to work around cpp bug
;;;
```

```
    shape->Rigid();
}

// Redraw displays the text label if the brush's the none brush, else
// it displays the graphic to demonstrate the brush's effect.

void BrushView::Redraw (Coord l, Coord b, Coord r, Coord t) {
    IBrush* brush = (IBrush*) brushindic->GetBrush();
    if (brush->None()) {
StateView::Redraw(l, b, r, t);
    } else {
output->ClearRect(canvas, l, b, r, t);
brushindic->Draw(canvas);
    }
}

// Resize scales the graphic to fit the canvas' size.

void BrushView::Resize () {
    StateView::Resize();
    float xmag = float(xmax) / PICXMAX;
    float hy = float(ymax) / 2;
    brushindic->SetTransformer(nil);
    brushindic->Scale(xmag, 1.);
    brushindic->Translate(0., hy);
}

// DrawingNameView just passes the drawing's name to StateView.

DrawingNameView::DrawingNameView (State* s) : (s, GetDrawingName(s)) {
}

// Update updates the view of the drawing's name.

void DrawingNameView::Update () {
    const char* drawingname = GetDrawingName(state);
    if (strcmp(drawingname, label) != 0) {
delete label;
label = strdup(drawingname);
Resize();
Draw();
    }
}

// Reconfig gives DrawingNameView's shape some stretchability to get
// more space if the HBox has room for it to stretch.

void DrawingNameView::Reconfig () {
    StateView::Reconfig();
    shape->hstretch = hfil;
}

// Resize left justifies the text label's position.

void DrawingNameView::Resize () {
    Font* font = output->GetFont();
    label_x = 0;
    label_y = (ymax - font->Height() + 1) / 2;
}

// GetDrawingName returns the drawing's name or a default label if it
// has no name.

const char* DrawingNameView::GetDrawingName (State* state) {
    const char* drawingname = state->GetDrawingName();
    return drawingname ? drawingname : "[unnamed drawing]";
}

// FontView passes the font's print name and size to StateView for its
// label.

FontView::FontView (State* s) : (s, GetPrintFontAndSize(s)) {
    background = nil;
}

// Free storage allocated for the background painter.
```

```
FontView::~FontView () {
   Unref(background);
}

// Update updates the view if the label or the current color changes.

void FontView::Update () {
   const char* name = GetPrintFontAndSize(state);
   Color* fgcolor = *state->GetFgColor();
   if (strcmp(name, label) != 0) {
delete label;
label = strdup(name);
output->SetColors(fgcolor, output->GetBgColor());
Resize();
Draw();
   } else if (output->GetFgColor() != fgcolor) {
output->SetColors(fgcolor, output->GetBgColor());
Draw();
   }
}

// Reconfig gives FontView's shape some stretchability to get more
// space if the HBox has room for it to stretch, creates a new painter
// to draw a gray background behind the label, and replaces output
// with a new painter to use a different color.

void FontView::Reconfig () {
   StateView::Reconfig();
   shape->hstretch = hfil;

   if (background == nil) {
background = new Painter(output);
background->Reference();
background->SetPattern(gray);
   }

   Color* fgcolor = *state->GetFgColor();
   Painter* copy = new Painter(output);
   copy->Reference();
   Unref(output);
   output = copy;
   output->SetColors(fgcolor, output->GetBgColor());
   output->FillBg(false);
}

// Redraw displays the graphic label over a gray background to make
// the label visible even if it uses the white color.

void FontView::Redraw (Coord l, Coord b, Coord r, Coord t) {
   background->FillRect(canvas, l, b, r, t);
   output->Text(canvas, label, label_x, label_y);
}

// GetPrintFontAndSize returns the font's print name and size.

const char* FontView::GetPrintFontAndSize (State* state) {
   IFont* f = state->GetFont();
   return f->GetPrintFontAndSize();
}

// GriddingView passes the grid's gridding on/off status to StateView.

GriddingView::GriddingView (State* s) : (s, GetGridding(s)) {
}

// Update updates the view of the grid's gridding on/off status.

void GriddingView::Update () {
   const char* gridding = GetGridding(state);
   if (strcmp(gridding, label) != 0) {
delete label;
label = strdup(gridding);
Draw();
   }
}
```

```
// GetGridding returns the status of the grid's gridding as a text
// string.

const char* GriddingView::GetGridding (State* state) {
   const char* gravity = nil;
   if (state->GetGridGravity()) {
gravity = "gridding on";
   } else {
gravity = "          ";
   }
   return gravity;
}

// MagnifView attaches itself to the DrawingView's perspective to get
// itself updated whenever the magnification changes.

MagnifView::MagnifView (State* s, Interactor* dwgview) : (s, GetMagnif(s)) {
   dwgview->GetPerspective()->Attach(this);
}

// Update updates the view of the current magnification.

void MagnifView::Update () {
   const char* magnif = GetMagnif(state);
   if (strcmp(magnif, label) != 0) {
delete label;
label = strdup(magnif);
Resize();
Draw();
   }
}

// Resize right justifies the text label's position.

void MagnifView::Resize () {
   Font* font = output->GetFont();
   label_x = xmax - font->Width(label) + 1;
   label_y = (ymax - font->Height() + 1) / 2;
}

// GetMagnif returns the drawing's magnification as a text string.

const char* MagnifView::GetMagnif (State* state) {
   static char mag[20];

   float magnif = state->GetMagnif();
   sprintf(mag, "  mag %.10gx ", magnif);
   return mag;
}

// ModifStatusView just passes the modification status to StateView.

ModifStatusView::ModifStatusView (State* s) : (s, GetModifStatus(s)) {
}

// Update updates the view of the current modification status.

void ModifStatusView::Update () {
   const char* modifstatus = GetModifStatus(state);
   if (strcmp(modifstatus, label) != 0) {
delete label;
label = strdup(modifstatus);
Draw();
   }
}

// GetModifStatus returns the drawing's modification status.

const char* ModifStatusView::GetModifStatus (State* state) {
   switch (state->GetModifStatus()) {
   case ReadOnly:
return "%";
   case Unmodified:
return " ";
   default:
```

```
return "*";
    }
}

// PatternView passes its none pattern label to StateView.

PatternView::PatternView (State* s) : (s, " N ") {
    patindic = nil;
}

// Free storage allocated for the fillrect painter.

PatternView::~PatternView () {
    Unref(patindic);
}

// Update updates the view if any of the colors or pattern changes.

void PatternView::Update () {
    Color* fgcolor = *state->GetFgColor();
    Color* bgcolor = *state->GetBgColor();
    Pattern* pattern = *state->GetPattern();
    if (patindic->GetFgColor() != fgcolor ||
patindic->GetBgColor() != bgcolor ||
patindic->GetPattern() != pattern)
    {
patindic->SetColors(fgcolor, bgcolor);
patindic->SetPattern(pattern);
Draw();
    }
}

// Reconfig creates the pattern indicator's painter.

void PatternView::Reconfig () {
    StateView::Reconfig();
    shape->Rigid();
    if (patindic == nil) {
Color* fgcolor = *state->GetFgColor();
Color* bgcolor = *state->GetBgColor();
Pattern* pattern = *state->GetPattern();
patindic = new Painter(output);
patindic->Reference();
patindic->SetColors(fgcolor, bgcolor);
patindic->SetPattern(pattern);
    }
}

// Redraw displays the text label if the pattern's the none pattern,
// else it displays the fillrect to demonstrate the pattern's effect.

void PatternView::Redraw (Coord l, Coord b, Coord r, Coord t) {
    if (state->GetPattern()->None()) {
StateView::Redraw(l, b, r, t);
    } else {
patindic->FillRect(canvas, l, b, r, t);
    }
}

MsgView::MsgView(State* s) : (s,GetMessage(s)) {
}

void MsgView::Update() {
const char* msg = GetMessage(state);
if (strcmp(msg,label) != 0) {
delete label;
label = strdup(msg);
Resize();
Draw();
    }
}

void MsgView::Reconfig() {
StateView::Reconfig();
shape->hstretch = hfil;
}
```

```
void MsgView::Resize() {
Font* font = output->GetFont();
label_x = 0;
label_y = (ymax - font->Height() + 1) / 2;
}

const char* MsgView::GetMessage(State* state) {
const char* msg = state->GetMessage();
return msg ? msg : " ";
}
```

textedit.h

```
#ifndef textedit_h
#define textedit_h

#include <InterViews/defs.h>

// Declare imported types.

class Canvas;
class Event;
class Painter;
class TextDisplay;
class TextBuffer;

// A TextEdit edits an array of lines of text.

class TextEdit {
public:

   TextEdit(const char* = 0, int = 0);
   ~TextEdit();

   const char* GetText(int&);
   char *GetString();

   void Redraw(Painter*, Canvas*, int lineHt, boolean redraw);
   void Grasp(Event&);
   boolean Editing(Event&);
   void Bounds(Coord& xmin, Coord& ymin, Coord& xmax, Coord& ymax);

protected:

   boolean HandleKey(char);

   void InsertCharacter(char);
   void DeleteCharacter(int);

   void InsertText(const char*, int);
   void DeleteText(int);

   void DeleteRestOfLine();
   void DeleteSelection();

   void BackwardCharacter(int = 1),   ForwardCharacter(int = 1);
   void BackwardLine(int = 1),        ForwardLine(int = 1);

   void BeginningOfLine(),            EndOfLine();

   void Select(int dot);
   void SelectMore(int mark);
   void SelectLine();
   void SelectWord();
   void Select(int dot, int mark);

   boolean Contains(Coord, Coord);
   int Locate(Coord, Coord);

protected:

   boolean selecting;
   int dot, mark;
   TextBuffer* text;
   TextDisplay* display;
```

```
    Painter* output;

};

#endif
```

textedit.c

```
#include "textedit.h"
#include "istring.h"
#include <InterViews/canvas.h>
#include <InterViews/event.h>
#include <InterViews/font.h>
#include <InterViews/painter.h>
#include <InterViews/textbuffer.h>
#include <InterViews/textdisplay.h>
#include <InterViews/transformer.h>
#include <bstring.h>
#include <ctype.h>

// Beware: only one instance of TextEdit can exist at any time!

static const int SBUFSIZE = 10000;
static char sbuf[SBUFSIZE];

TextEdit::TextEdit (const char* sample, int samplen) {
    selecting = false;
    dot = mark = 0;
    text = new TextBuffer(sbuf, 0, SBUFSIZE);
    display = new TextDisplay(true);
    output = nil;

    text->Insert(0, sample, samplen);
    int lines = text->Height();
    for (int i = 0; i < lines; ++i) {
        int bol = text->LineIndex(i);
        int eol = text->EndOfLine(bol);
        display->ReplaceText(i, text->Text(bol, eol), eol - bol);
    }
}

TextEdit::~TextEdit () {
    delete text;
    delete display;
}

const char* TextEdit::GetText (int& size) {
    size = text->Length();
    return text->Text();
}

char *TextEdit::GetString() {
   int len;
   const char *str = GetText(len);
   char *rval = new char[len + 1];
   strncpy(rval,str,len);
   rval[len] = 0;
   return rval;
}



void TextEdit::Redraw (
    Painter* output, Canvas* canvas, int lineHt, boolean redraw
) {
    TextEdit::output = output;
    display->Draw(output, canvas);

    display->LineHeight(lineHt);
    Coord l = 0;
    Coord b = 0 - display->Height();
    Coord r = 0 + display->Width();
    Coord t = 0;
    display->Resize(l, b, r, t);
    if (redraw) {
display->Redraw(l, b, r, t);
```

```
    }
}

void TextEdit::Grasp (Event& e) {
    Select(Locate(e.x, e.y));
    selecting = true;
    e.eventType = MotionEvent;
}

boolean TextEdit::Editing (Event& e) {
    boolean editing = true;

    if (e.eventType == KeyEvent && e.len > 0) {
        editing = HandleKey(e.keystring[0]);
    } else if (e.eventType == MotionEvent && selecting) {
        SelectMore(Locate(e.x, e.y));
    } else if (e.eventType == DownEvent) {
        if (e.shift) {
            SelectMore(Locate(e.x, e.y));
            selecting = true;
        } else if (Contains(e.x, e.y)) {
            Select(Locate(e.x, e.y));
            selecting = true;
        } else {
            editing = false;
        }
    } else if (e.eventType == UpEvent) {
        selecting = false;
    }

    if (!editing) {
Select(dot);
        display->CaretStyle(NoCaret);
    }
    return editing;
}

void TextEdit::Bounds (Coord& xmin, Coord& ymin, Coord& xmax, Coord& ymax) {
    Transformer* t = output->GetTransformer();

    display->Bounds(xmin, ymin, xmax, ymax);
    if (t != nil) {
        t->TransformRect(xmin, ymin, xmax, ymax);
    }
}

boolean TextEdit::HandleKey (char c) {
    boolean editing = true;

    switch (c) {
        case '\001' /* ^A  */: BeginningOfLine(); break;
        case '\005' /* ^E  */: EndOfLine(); break;
        case '\006' /* ^F  */: ForwardCharacter(1); break;
        case '\002' /* ^B  */: BackwardCharacter(1); break;
        case '\016' /* ^N  */: ForwardLine(1); break;
        case '\020' /* ^P  */: BackwardLine(1); break;
        case '\013' /* ^K  */: DeleteRestOfLine(); break;
        case '\004' /* ^D  */: DeleteCharacter(1); break;
        case '\010' /* ^H  */: DeleteCharacter(-1); break;
        case '\177' /* DEL */: DeleteCharacter(-1); break;
        case '\011' /* TAB */: InsertCharacter('\t'); break;
        case '\027' /* ^W  */: SelectWord(); break;
        case '\025' /* ^U  */: SelectLine(); break;
        case '\015' /* RET */: InsertCharacter('\n'); break;
        case '\033' /* ESC */: editing = false; break;
        default:
            if (!iscntrl(c)) {
                InsertCharacter(c);
            }
            break;
    }
    return editing;
}

void TextEdit::InsertCharacter (char c) {
    DeleteSelection();
```

```
    InsertText(&c, 1);
}

void TextEdit::DeleteCharacter (int count) {
    if (dot != mark) {
        DeleteSelection();
    } else {
        DeleteText(count);
    }
}

void TextEdit::InsertText (const char* s, int count) {
    count = text->Insert(dot, s, count);
    int sline = text->LineNumber(dot);
    int fline = text->LineNumber(dot + count);
    if (sline == fline) {
        int offset = text->LineOffset(dot);
        display->InsertText(sline, offset, text->Text(dot), count);
    } else {
        display->InsertLinesAfter(sline, fline-sline);
        for (int i = sline; i <= fline; ++i) {
            int bol = text->BeginningOfLine(text->LineIndex(i));
            int eol = text->EndOfLine(bol);
            display->ReplaceText(i, text->Text(bol, eol), eol-bol);
        }
    }
    Select(dot + count);
}

void TextEdit::DeleteText (int count) {
    int d = dot;
    int c = count;
    while (c > 0) {
        d = text->NextCharacter(d);
        --c;
    }
    while (c < 0) {
        dot = text->PreviousCharacter(dot);
        ++c;
    }
    count = d - dot;
    int sline = text->LineNumber(dot);
    int fline = text->LineNumber(d);
    text->Delete(dot, count);
    if (sline == fline) {
        int offset = text->LineOffset(dot);
        display->DeleteText(sline, offset, count);
    } else {
        int bol = text->BeginningOfLine(dot);
        int eol = text->EndOfLine(dot);
        display->DeleteLinesAfter(sline, fline-sline);
        display->ReplaceText(sline, text->Text(bol, eol), eol-bol);
    }
    Select(dot);
}

void TextEdit::DeleteRestOfLine () {
    if (dot == mark) {
int bol = text->BeginningOfLine(dot);
if (dot == bol) {
    Select(dot, text->BeginningOfNextLine(dot));
} else {
    Select(dot, text->EndOfLine(dot));
}
    }
    DeleteSelection();
}

void TextEdit::DeleteSelection () {
    if (dot != mark) {
        DeleteText(mark - dot);
    }
}

void TextEdit::BeginningOfLine () {
    if (dot != mark) {
```

```
    Select(min(mark, dot));
  } else {
    Select(text->BeginningOfLine(dot));
  }
}

void TextEdit::EndOfLine () {
  if (dot != mark) {
    Select(max(mark, dot));
  } else {
    Select(text->EndOfLine(dot));
  }
}

void TextEdit::ForwardCharacter (int count) {
  if (dot != mark) {
    Select(max(mark, dot));
  } else {
    int d = dot;
    while (count > 0) {
      d = text->NextCharacter(d);
      --count;
    }
    Select(d);
  }
}

void TextEdit::BackwardCharacter (int count) {
  if (dot != mark) {
    Select(min(mark, dot));
  } else {
    int d = dot;
    while (count > 0) {
      d = text->PreviousCharacter(d);
      --count;
    }
    Select(d);
  }
}

void TextEdit::ForwardLine (int count) {
  if (dot != mark) {
    Select(max(mark, dot));
  } else {
    int d = dot;
    while (count > 0) {
      d = text->BeginningOfNextLine(d);
      --count;
    }
    Select(d);
  }
}

void TextEdit::BackwardLine (int count) {
  if (dot != mark) {
    Select(min(mark, dot));
  } else {
    int d = dot;
    while (count > 0) {
      d = text->BeginningOfLine(text->EndOfPreviousLine(d));
      --count;
    }
    Select(d);
  }
}

void TextEdit::Select (int d) {
  Select(d, d);
}

void TextEdit::SelectMore (int m) {
  Select(dot, m);
}

void TextEdit::SelectLine () {
  Select(text->BeginningOfLine(dot), text->BeginningOfNextLine(dot));
```

```
}

void TextEdit::SelectWord () {
  int left = min(mark, dot);
  int right = max(mark, dot);
  Select(text->BeginningOfWord(text->PreviousCharacter(left)), right);
}

void TextEdit::Select (int d, int m) {
  int oldl = min(dot, mark);
  int oldr = max(dot, mark);
  int newl = min(d, m);
  int newr = max(d, m);
  if (oldl == oldr && newl != newr) {
    display->CaretStyle(NoCaret);
  }
  if (newr < oldl || newl > oldr) {
    if (oldr > oldl) {
      display->RemoveStyle(
        text->LineNumber(oldl), text->LineOffset(oldl),
        text->LineNumber(oldr-1), text->LineOffset(oldr-1),
        Reversed
      );
    }
    if (newr > newl) {
      display->AddStyle(
        text->LineNumber(newl), text->LineOffset(newl),
        text->LineNumber(newr-1), text->LineOffset(newr-1),
        Reversed
      );
    }
  } else {
    if (newl < oldl) {
      display->AddStyle(
        text->LineNumber(newl), text->LineOffset(newl),
        text->LineNumber(oldl-1), text->LineOffset(oldl-1),
        Reversed
      );
    } else if (newl > oldl) {
      display->RemoveStyle(
        text->LineNumber(oldl), text->LineOffset(oldl),
        text->LineNumber(newl-1), text->LineOffset(newl-1),
        Reversed
      );
    }
    if (newr > oldr) {
      display->AddStyle(
        text->LineNumber(oldr), text->LineOffset(oldr),
        text->LineNumber(newr-1), text->LineOffset(newr-1),
        Reversed
      );
    } else if (newr < oldr) {
      display->RemoveStyle(
        text->LineNumber(newr), text->LineOffset(newr),
        text->LineNumber(oldr-1), text->LineOffset(oldr-1),
        Reversed
      );
    }
  }
  if (oldl != oldr && newl == newr) {
    display->CaretStyle(BarCaret);
  }
  if (newl == newr) {
    display->Caret(text->LineNumber(newl), text->LineOffset(newl));
  }
  dot = d;
  mark = m;
}

boolean TextEdit::Contains (Coord x, Coord y) {
  Transformer tr(output->GetTransformer());
  tr.InvTransform(x, y);
  int line = display->LineNumber(y);
  int index = display->LineIndex(line, x);

  return
```

```
        x >= display->Left(line, text->BeginningOfLine(index)) &&
        x <= display->Right(line, text->EndOfLine(index)) &&
        y >= display->Base(line) &&
        y <= display->Top(line);
}

int TextEdit::Locate (Coord x, Coord y) {
    Transformer tr(output->GetTransformer());
    tr.InvTransform(x, y);
    int line = display->LineNumber(y);
    int index = display->LineIndex(line, x);
    int l = text->LineIndex(line);
    int i = 0;
    while (i < index) {
        l = text->NextCharacter(l);
        i += 1;
    }
    return l;
}
```

tools.h

```
#ifndef tools_h
#define tools_h

#include "panel.h"

// Declare imported types.

class Editor;
class MapKey;

// A Tools displays several drawing tools to choose from.

class Tools : public Panel {
public:

    Tools(Editor*, MapKey*);

    void Handle(Event&);

protected:

    void Init(Editor*, MapKey*);
    void Reconfig();

};

#endif
```

tools.c

```
#include "editor.h"
#include "keystrokes.h"
#include "mapkey.h"
#include "tools.h"
#include <InterViews/box.h>
#include <InterViews/event.h>
#include <InterViews/painter.h>
#include <InterViews/shape.h>
#include <InterViews/Std/string.h>

// An IdrawTool enters itself into the MapKey so Idraw can send
// a KeyEvent to the right IdrawTool.

class IdrawTool : public PanelItem {
public:
    IdrawTool(Panel*, const char*, char, Editor*, MapKey*);
protected:
    Editor* editor;// handles drawing and editing operations
};
```

```
// IdrawTool stores the editor pointer and enters itself and its
// associated character into the MapKey.

IdrawTool::IdrawTool (Panel* p, const char* n, char c, Editor* e,
MapKey* mapkey) : PanelItem(p, n, mapkey->ToStr(c), c, e)
{
   editor = e;
   mapkey->Enter(this, c);
}


// A SelectTool selects a set of Selections.


class SelectTool : public IdrawTool {
public:
   SelectTool (Panel* p, Editor* e, MapKey* mk)
    : IdrawTool(p, "Select", SELECTCHAR, e, mk) { }
   void SetMessage()
   {
     strcpy(msg, "pick object with LMB or hold down button to draw rectangle");
     strcat(msg, " around more than 1 object");
     editor->ResetMessage(msg);
   }

void Perform (Event& e) {
editor->HandleSelect(e);
   }
};



// A MoveTool moves a set of Selections.

class MoveTool : public IdrawTool {
public:
   MoveTool (Panel* p, Editor* e, MapKey* mk)
       : IdrawTool(p, "Move", MOVECHAR, e, mk) { }
   void SetMessage()
    {
strcpy(msg, "pick object with LMB");
      editor->ResetMessage(msg);
   }
   void Perform (Event& e)
   {
editor->HandleMove(e);
   }
};


// A DecomposeTool will open new graphic editor for lower level of DFD

class DecomposeTool : public IdrawTool {
public:
DecomposeTool (Panel* p, Editor* e, MapKey* mk)
: IdrawTool(p, "Implement/Decompose", DECOMPOSECHAR, e, mk) { }
void SetMessage() {
strcpy(msg,
 "pick operator with LMB");
editor->ResetMessage(msg);
   }
void Perform (Event& e) {
editor->HandleDecompose(e);
   }
};

// A CommentTool draws some text.

class CommentTool : public IdrawTool {
public:
   CommentTool (Panel* p, Editor* e, MapKey* mk)
   : IdrawTool(p, "Title/Comment", COMMENTCHAR, e, mk) { }
void SetMessage() {
strcpy(msg,
```

```
 "click outside of objects to add text");
 editor->ResetMessage(msg);
   }
   void Perform (Event& e) {
 editor->HandleText(e);
   }
 };


 // A LabelTool draws some text for one of the components of the DFD

 class LabelTool : public IdrawTool {
 public:
   LabelTool (Panel* p, Editor* e, MapKey* mk)
     : IdrawTool(p, "Label", LABELCHAR, e, mk) { }
   void SetMessage() {
     strcpy(msg,"pick object with LMB, add text, then click outside objects");
     editor->ResetMessage(msg);
   }
   void Perform (Event& e) {
     editor->HandleLabel(e);
   }
 };


 // A TriggerIfTool draws some text for one of the components of the DFD

 class IfTool : public IdrawTool {
 public:
   IfTool (Panel* p, Editor* e, MapKey* mk)
     : IdrawTool(p, "Trigger If", IFCHAR, e, mk) { }
   void SetMessage() {
     strcpy(msg,"pick object with LMB, add text, then click outside objects");
     editor->ResetMessage(msg);
   }
   void Perform (Event& e) {
     editor->HandleIf(e);
   }
 };

 // A TriggerBYALLTool draws some text for one of the components of the DFD

 class By_AllTool : public IdrawTool {
 public:
   By_AllTool (Panel* p, Editor* e, MapKey* mk)
     : IdrawTool(p, "Trigger By All", BYALLCHAR, e, mk) { }
   void SetMessage() {
     strcpy(msg,"pick object with LMB, add text, then click outside objects");
     editor->ResetMessage(msg);
   }
   void Perform (Event& e) {
     //editor->HandleBy_All(e);
   }
 };

 // A TriggerBY_SOMETool draws some text for one of the components of the DFD

 class By_SomeTool : public IdrawTool {
 public:
   By_SomeTool (Panel* p, Editor* e, MapKey* mk)
     : IdrawTool(p, "Trigger By Some", BYSOMECHAR, e, mk) { }
   void SetMessage() {
     strcpy(msg,"pick object with LMB, add text, then click outside objects");
     editor->ResetMessage(msg);
   }
   void Perform (Event& e) {
    // editor->HandleBy_Some(e);
   }
 };

 // A HourTool adds the maximu execution of the operator to the drawing

 class HourTool : public IdrawTool
  {
    public:
      HourTool (Panel* p, Editor* e, MapKey* mk)
```

```
    : IdrawTool(p, "HOUR " , HOURCHAR, e, mk) { }
    void SetMessage()
     {
      strcpy(msg,
         "pick operator with LMB, add text, then click outside object");
      editor->ResetMessage(msg);
     }
    void Perform(Event& e)
     {
      editor->HandleMET(e , " hours");
     }
};

// A HourLTool adds the maximu execution of the STREAM to the drawing

class HourLTool : public IdrawTool
 {
  public:
    HourLTool (Panel* p, Editor* e, MapKey* mk)
    : IdrawTool(p, "HOUR " , LHOURCHAR, e, mk) { }
    void SetMessage()
     {
      strcpy(msg,
         "pick operator with LMB, add text, then click outside object");
      editor->ResetMessage(msg);
     }
    void Perform(Event& e)
     {
      editor->HandleLatency(e , " hours");
     }
};


// A MinuteTool adds the maximu execution of the operator to the drawing

class MinuteTool : public IdrawTool
 {
  public:
    MinuteTool (Panel* p, Editor* e, MapKey* mk)
    : IdrawTool(p, "Minute " , MINUTECHAR, e, mk) { }
    void SetMessage()
     {
      strcpy(msg,
         "pick operator with LMB, add text, then click outside object");
      editor->ResetMessage(msg);
     }
    void Perform(Event& e)
     {
      editor->HandleMET(e , " min");
     }
};


// A MinuteLTool adds the maximu execution of the STREAM to the drawing

class MinuteLTool : public IdrawTool
 {
  public:
    MinuteLTool (Panel* p, Editor* e, MapKey* mk)
    : IdrawTool(p, "Minute " ,LMINUTECHAR, e, mk) { }
    void SetMessage()
     {
      strcpy(msg,
         "pick operator with LMB, add text, then click outside object");
      editor->ResetMessage(msg);
     }
    void Perform(Event& e)
     {
      editor->HandleLatency(e , " min");
     }
};



// A SecTool adds the maximu execution of the operator to the drawing
```

```
class SecTool : public IdrawTool
 {
  public:
    SecTool (Panel* p, Editor* e, MapKey* mk)
    : IdrawTool(p, "Second " , SECONDCHAR, e, mk) { }
    void SetMessage()
     {
      strcpy(msg,
         "pick operator with LMB, add text, then click outside object");
      editor->ResetMessage(msg);
     }
    void Perform(Event& e)
     {
      editor->HandleMET(e , " sec");
     }
};


// A SecLTool adds the maximu execution of the STREAM to the drawing

class SecLTool : public IdrawTool
 {
  public:
    SecLTool (Panel* p, Editor* e, MapKey* mk)
    : IdrawTool(p, "Second " , LSECONDCHAR, e, mk) { }
    void SetMessage()
     {
      strcpy(msg,
         "pick operator with LMB, add text, then click outside object");
      editor->ResetMessage(msg);
     }
    void Perform(Event& e)
     {
      editor->HandleLatency(e , " sec");
     }
};


// A MilSecTool adds the maximu execution of the operator  to the drawing

class MilSecTool : public IdrawTool
 {
  public:
    MilSecTool (Panel* p, Editor* e, MapKey* mk)
    : IdrawTool(p, "MilliSecond " , MILSECCHAR, e, mk) { }
    void SetMessage()
     {
      strcpy(msg,
         "pick operator with LMB, add text, then click outside object");
      editor->ResetMessage(msg);
     }
    void Perform(Event& e)
     {
      editor->HandleMET(e , " ms");
     }
};


// A MilSecLTool adds the maximu execution of the STREAM to the drawing

class MilSecLTool : public IdrawTool
 {
  public:
    MilSecLTool (Panel* p, Editor* e, MapKey* mk)
    : IdrawTool(p, "MilliSecond " , LMILSECCHAR, e, mk) { }
    void SetMessage()
     {
      strcpy(msg,
         "pick operator with LMB, add text, then click outside object");
      editor->ResetMessage(msg);
     }
    void Perform(Event& e)
     {
      editor->HandleLatency(e , " ms");
     }
```

```
};

// A MicroSecTool adds the maximu execution of the operator to the drawing

class MicroSecTool : public IdrawTool
 {
  public:
    MicroSecTool (Panel* p, Editor* e, MapKey* mk)
    : IdrawTool(p, "us " , MICSECCHAR, e, mk) { }
    void SetMessage()
     {
      strcpy(msg,
         "pick operator with LMB, add text, then click outside object");
      editor->ResetMessage(msg);
     }
    void Perform(Event& e)
     {
      editor->HandleMET(e , " microsec");
     }
};


// A MicroSecLTool adds the maximu execution of the STREAM to the drawing

class MicroSecLTool : public IdrawTool
 {
  public:
    MicroSecLTool (Panel* p, Editor* e, MapKey* mk)
    : IdrawTool(p, "us " , LMICSECCHAR, e, mk) { }
    void SetMessage()
     {
      strcpy(msg,
         "pick operator with LMB, add text, then click outside object");
      editor->ResetMessage(msg);
     }
    void Perform(Event& e)
     {
      editor->HandleLatency(e , " microsec");
     }
};



// A BSplineTool draws an open B-spline.

class BSplineTool : public IdrawTool {
public:
   BSplineTool (Panel* p, Editor* e, MapKey* mk)
   : IdrawTool(p, "", BSPLINECHAR, e, mk) { }
void SetMessage() {
strcpy(msg,
 "click LMB at each spot to place segment of spline");
editor->ResetMessage(msg);
   }
   void Perform (Event& e) {
editor->HandleBSpline(e);
   }
protected:
   void Redraw (Coord l, Coord b, Coord r, Coord t) {
IdrawTool::Redraw(l, b, r, t);
const int N = 4;
Coord x[N];
Coord y[N];
x[0] = offx + side * 1/5;
y[0] = offy + side * 4/5;
x[1] = offx + side * 1/2;
y[1] = offy + side * 4/5;
x[2] = offx + side * 1/2;
y[2] = offy + side * 1/5;
x[3] = offx + side * 4/5;
y[3] = offy + side * 1/5;
output->BSpline(canvas, x, y, N);
   }
};

// An EllipseTool draws an ellipse.
```

```
class EllipseTool : public IdrawTool {
public:
   EllipseTool (Panel* p, Editor* e, MapKey* mk)
   : IdrawTool(p, "", ELLIPSECHAR, e, mk) { }
void SetMessage() {
strcpy(msg,
 "click with LMB to draw operator centered at that point");
editor->ResetMessage(msg);
   }
   void Perform (Event& e) {
editor->HandleEllipse(e);
   }
protected:
   void Redraw (Coord l, Coord b, Coord r, Coord t) {
IdrawTool::Redraw(l, b, r, t);
Coord x0 = offx + side * 1/2;
Coord y0 = offy + side * 1/2;
Coord xradius = side * 1/3 + side * 1/16;
Coord yradius = side * 1/3 - side * 1/16;
output->Ellipse(canvas, x0, y0, xradius, yradius);
   }
};

// A RectTool draws a rectangle.

class RectTool : public IdrawTool {
public:
   RectTool (Panel* p, Editor* e, MapKey* mk)
   : IdrawTool(p, "", RECTCHAR, e, mk) { }
  void SetMessage() {
strcpy(msg,
 "click with LMB to draw terminator centered at that point");
editor->ResetMessage(msg);
   }
   void Perform (Event& e) {
      editor->HandleRect(e);
   }
protected:
   void Redraw (Coord l, Coord b, Coord r, Coord t) {
      IdrawTool::Redraw(l, b, r, t);
      Coord x0 = offx + side * 1/5;
      Coord y0 = offy + side * 1/5;
      Coord x1 = offx + side * 4/5;
      Coord y1 = offy + side * 4/5;
      output->Rect(canvas, x0, y0, x1, y1);
   }
};


// Tools creates its tools.

Tools::Tools (Editor* e, MapKey* mk) {
   Init(e, mk);
}

// Handle tells one of the tools to perform its function if a
// DownEvent occurs.

void Tools::Handle (Event& e) {
   switch (e.eventType) {
   case DownEvent:
switch (e.button) {
case LEFTMOUSE:
   PerformCurrentFunction(e);
   break;
case MIDDLEMOUSE:
   PerformTemporaryFunction(e, MOVECHAR);
   break;
case RIGHTMOUSE:
   PerformTemporaryFunction(e, SELECTCHAR);
   break;
default:
   break;
}
   default:
```

```
break;
    }
}

// Init creates the tools, lays them together, and inserts them.

void Tools::Init (Editor* e, MapKey* mk) {
    PanelItem* first = new SelectTool(this, e, mk);

    VBox* tools = new VBox;
    tools->Insert(first);

    //tools -> Insert(new PropertyTool(this, e, mk ));
    tools->Insert(new MoveTool(this, e, mk));

    // tools->Insert(new ModifyTool(this, e, mk));
    //tools->Insert(new SpecifyTool(this, e, mk));

    tools->Insert(new DecomposeTool(this, e, mk));
    tools->Insert(new CommentTool(this, e, mk));

     Enter(new HourTool    ( this, e, mk), HOURCHAR );
     Enter(new MinuteTool  ( this, e, mk), MINUTECHAR);
     Enter(new SecTool     ( this, e, mk), SECONDCHAR);
     Enter(new MilSecTool  ( this, e, mk), MILSECCHAR);
     Enter(new MicroSecTool ( this, e, mk), MICSECCHAR);


     Enter(new HourLTool    ( this, e, mk), LHOURCHAR );
     Enter(new MinuteLTool  ( this, e, mk), LMINUTECHAR);
     Enter(new SecLTool     ( this, e, mk), LSECONDCHAR);
     Enter(new MilSecLTool  ( this, e, mk), LMILSECCHAR);
     Enter(new MicroSecLTool ( this, e, mk), LMICSECCHAR);


//  This part is for Trigger If/By_All/By_Some

     Enter ( new IfTool      ( this, e, mk), IFCHAR);
     Enter ( new By_AllTool  ( this, e, mk), BYALLCHAR);
     Enter ( new By_SomeTool  ( this, e, mk), BYSOMECHAR);


    tools->Insert(new BSplineTool(this, e, mk));
    tools->Insert(new EllipseTool(this, e, mk));
    tools->Insert(new RectTool(this, e, mk));
//   tools->Insert(new LabelTool(this, e, mk));
    Enter(new LabelTool(this, e, mk), LABELCHAR);

    Insert(tools);
// Reconfig makes Tools's shape unstretchable but shrinkable.

void Tools::Reconfig () {
    Panel::Reconfig();
    shape->Rigid(0, 0, vfil, 0);
}
```

LIST OF REFERENCES

1 [BROC94] Brockett, Jim, The Computer-Aided Prototyping System Tutorial, Computer Science Department, Naval Postgraduate School, California, July, 1994.

2 [CUMM90] Cummings, Mary Ann, The Development of User Interface Tools for the Computer_Aided Prototyping System, M. S. Thesis, Naval Postgraduate School, Monterey, California, December, 1990.

3 [DIXO92] Dixon, R., The Design and Implementation of a User Interface for the Computer-Aided Prototyping System, M. S. Thesis, Naval Postgraduate School, Monterey, California, September, 1992.

4 [HELL91] Heller, Dan, Motif Programming Manual for OSF/Motif Version 1.1, O'Reilly and Associates, September, 1991.

5 [LUQI88] Luqi and M. Ketabchi, A Computer_Aided Prototyping System, IEEE Software, March 1988, 66-72.

6 [LBY 88] Luqi, Berzins, V. and Yeh, R., A Prototyping Language for Real_Time Software, IEEE Transactions on Software Engineering, October, 1988, 1409-1423.

7 [LUQI89] Luqi, Handling Timing Constraints in Rapid Prototyping, Proceedings of the Twenty-Second Annual Hawaii Conference on Systems Science, January, 1989, 417-424.

8 [LUQI92] Luqi, Computer_Aided Prototyping for a Command_and_Control System Using CAPS, IEEE software, January, 1992, 56-67.

9 [STAN91] Stanford University, InterViews Reference Manual, Version 3.0, December, 1991.

10 [VLIS89] Vlissides, John, and Linton, Mark, Idraw/Unidraw: A Framework for Building Domain-Specific Graphical Editors, Proceedings of ACM SIGGRAPH/ SIGCHI, November, 1989.

INITIAL DISTRIBUTION LIST

1.	Defense Technical Information Center Cameron Station Alexandria, VA 22304-6145	2
2.	Dudley Knox Library Code 052 Naval Postgraduate School Monterey, CA 93943-5101	2
3.	Chairman, Code CS Computer Science Department Naval Postgraduate School Monterey, CA 93943-5000	1
4.	Dr Mantak Shing, Code CS/Mz Computer Science Department Naval Postgraduate School Monterey, CA 93943-5000	2
5.	Lt. Colonel David A. Gaitros, Code CS/Gi Computer Science Department Naval Postgraduate School Monterey, CA 93943-5000	1
6.	Dr. Luqi, Code CS/Lq Computer Science Department Naval Postgraduate School Monterey, CA 93943-5000	1
7.	Dr. Valdis Berzins, Code CS/Be Computer Science Department Naval Postgraduate School Monterey, CA 93943-5000	1
8.	United States Army Computer Science School ATTN: ATZH-SS Fort Gordon, GA 30905	2

9. United States Army Signal Corps and Fort Gordon 2
Conrad Technical Library
BLDG 29807
Fort Gordon, GA 30905-5081

10. Captain Mehdi E. Rowshanaee 2
US Army Computer Science School
Attn: ATZH-SS
Fort Gordon, GA 30905